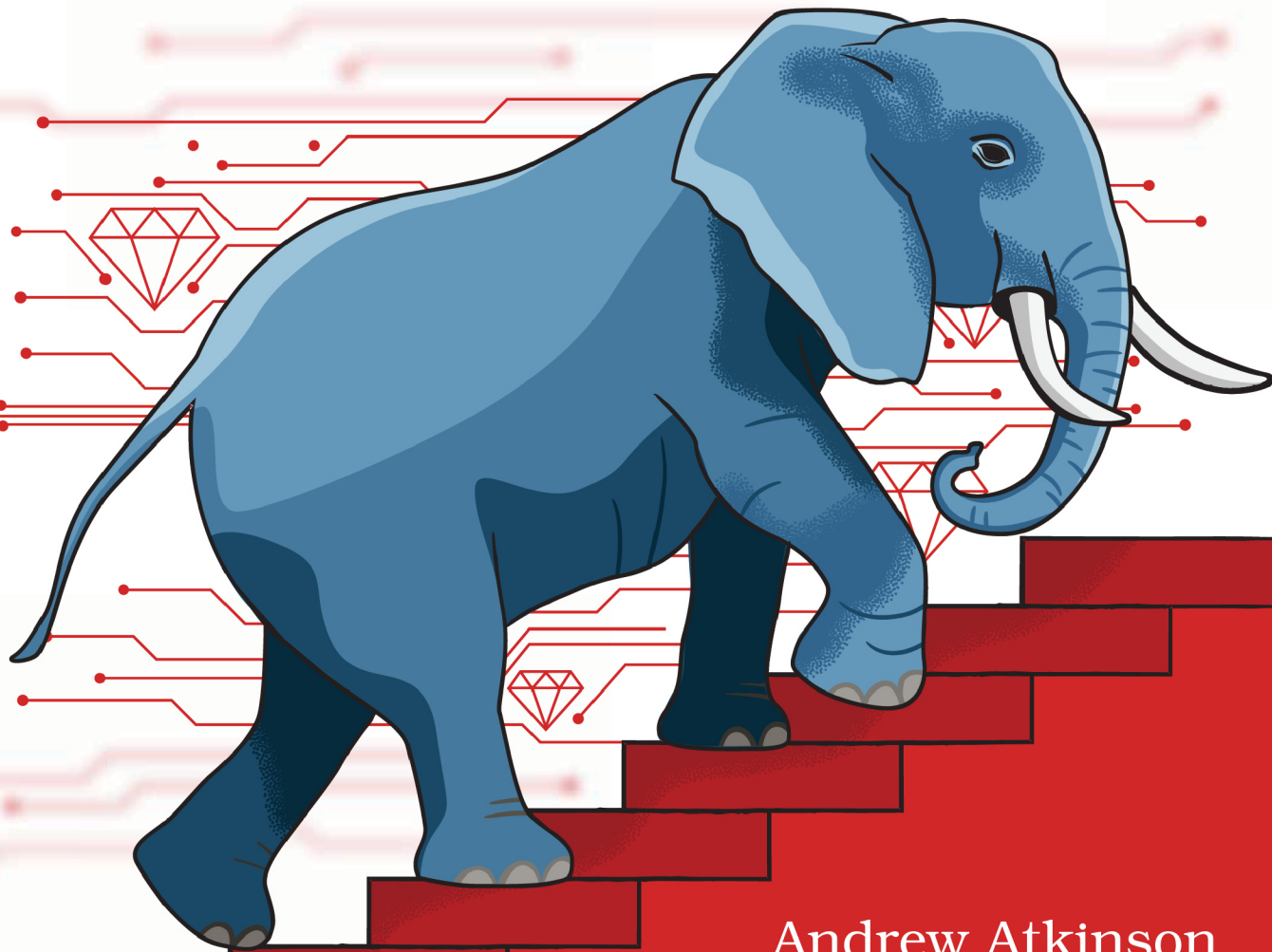


The
Pragmatic
Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable
Database Applications



Andrew Atkinson
edited by Don N. Hagist



Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at pragprog.com/my_account. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at pragprog.com/titles/aapsql/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

The Pragmatic Bookshelf

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

The Pragmatic Bookshelf

Dallas, Texas



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 979-8-88865-038-7

Encoded using the finest acid-free high-entropy binary digits.

Book version: B2.0—September 13, 2023

Contents

Change History	ix
Preface	xi

Part I — Getting Started

1. An App to Get You Started	3
What Is Rideshare?	3
Active Record Schema Management Refresher	4
Installing Rideshare	6
Working With PostgreSQL Locally	10
Learning PostgreSQL Terminology	11
Learning SQL Terminology	12
Ruby on Rails Terminology	13
Conventions Used In This Book	14
SQL Conventions	14
Rails Conventions	14
You're Ready	15

Part II — Design and Build

2. Administration Basics	19
Learning Meta Commands In psql With Rideshare	19
Modifying Your PostgreSQL Config File	22
Getting Started With Query Observability	24
Glancing At Current Lock Behavior	25
Generating Fake Data For Experiments	27
Creating Indexes Using SQL	28

	Rolling Back Schema Modifications	29
	Exploring and Experimenting Safely In Production	30
3.	Building a Performance Testing Database	31
	Downloading Production Data Dumps	32
	Replacement Values That Are Statistically Similar	34
	Tracking Columns With Sensitive Information	36
	Comparing Direct Updates and Table Copying Strategies	37
	Starting An Email Scrubber Function	38
	Implementing the Scrub Email Function	40
	Understanding Table Copying Trade-Offs	43
	Speeding Up Inserts For Table Copying	45
	Using Direct Updates For Text Replacement	50
	Performing Database Maintenance	50
	Performing Modifications In Batches	52
	What's Next For Your Performance Database	53
4.	Data Correctness and Consistency	55
	Multiple Column Uniqueness	56
	Fixing Constraint Violations	57
	Enforcing Relationships With Foreign Keys	58
	The Versatile Check Constraint	60
	Deferring Constraint Checks	62
	Preventing Overlaps With An Exclusion Constraint	63
	Creating Active Record Custom Validators	65
	Significant Casing And Unique Constraints	67
	Storing Transformations In Generated Columns	69
	Constraining Values With Database Enums	70
	Sharing Domains Between Tables	72
	Automating Consistency Checks In Development	74
Part III — Operate and Grow		
5.	Modifying Busy Databases Without Downtime	77
	Identifying Dangerous Migrations	78
	Learning From Unsafe Migrations	79
	Learning To Use CONCURRENTLY By Default	81
	Adopting A Migration Safety Check Process	82
	Exploring Strong Migrations Features	82
	Locking, Blocking, and Concurrency Refresher	85

Preventing Excessive Queueing With A Lock Timeout	87
Exploring Lock Type Queues	88
Setting Statement Timeout	89
Avoiding Schema Cache Errors	90
Backfilling Large Tables Without Downtime	90
Backfilling And Double Writing	91
Separating Reads and Writes for Backfills	92
Specialized Tables For Backfills	92
Practicing Backfilling Techniques	93
Wrapping Up	94
6. Optimizing Active Record	97
Preferring Active Record Over SQL	97
Query Logs To Connect SQL to App Code	98
Common Active Record Problems	99
Use Eager Loading To Reduce Queries	102
Eager Loading With .includes()	102
Prefer Strict Loading Over Lazy Loading	103
Optimizing Individual Active Record Queries	104
Save A SELECT With A RETURNING	105
Bounding Query Results Using LIMIT	106
Advanced Query Support In Active Record	107
Using Common Table Expressions (CTE)	108
Introducing Database Views for Rideshare	110
Creating the Search Result Model With Scenic	112
Improving Performance With Materialized Views	113
Reducing Queries With Active Record Caches	114
Prepared Statements With Active Record	115
Eliminating Slow Count Queries With Counter Caches	116
Performing Aggregations In the Database	118
Reduced Object Allocations With SQL In Active Record	119
Wrapping Up	120
7. Improving Query Performance	121
Logging Slow Queries With Active Support Notifications	122
Capture Query Statistics In Your Database	123
Rideshare Query Statistics	124
Introducing PgHero As a Performance Dashboard	125
Analyzing Query Execution Plans	127
Finding Missing Indexes	130
Logging Slow Queries	131

Automatically Gathering Execution Plans	132
Performing Maintenance First	133
Interpreting Index Scan Execution Plan Info	134
EXPLAIN Scan Nodes and Bitmap Scans	135
Adding Query Boundaries With Filtering and LIMIT	136
Performing Fast COUNT() Queries	137
Using Code and SQL Analysis Tools	139
Wrapping Up	139
8. Optimized Indexes For Fast Retrieval	141
Generating Data for Experiments	142
Transforming Ruby to SQL	145
Why Are My Indexes Not Being Used?	146
Using Single Column And Multiple Column Indexes	146
Understanding Index Column Ordering	147
Indexing Boolean Columns	148
Transform Values with an Expression Index	149
Using GIN Indexes with JSON	150
Using GIN Indexes for Full Text Search	154
Using Partial Indexes	157
Using BRIN Indexes	159
Using Indexes In Less Common Ways	162
Using Covering Indexes	163
9. High Impact Database Maintenance	167
Basics of Autovacuum	168
Tuning Autovacuum Parameters	169
Rebuilding Indexes Without Downtime	171
Running Manual Vacuums	172
Simulating Bloat and Understanding Impact	173
Removing Unused Indexes	175
Removing Duplicate And Overlapping Indexes	176
Removing Indexes On Insert Only Tables	178
Scheduling Jobs Using pg_cron	178
Monitoring pg_cron Scheduled Jobs	180
Conducting Maintenance Tune-Ups	181
10. Handling Errors From Increased Concurrency	183
Monitoring Database Connections	184
Managing Idle Connections	186
Setting Active Record Pool Size	187

Using the Active Record Connection Pool	187
Running Out Of Connections	188
Working With PgBouncer	191
Choosing A PgBouncer Pooling Mode	193
Identifying Connection Errors and Problems	194
More Lock Monitoring With <code>pg_locks</code>	196
Monitoring Row Locks	196
Finding Lock Conflicts	197
Using PgBadger For Lock Analysis	198
Active Record Optimistic Locking	198
Using Advisory Locks	199
Wrapping Up	200

Part IV — Optimize and Scale

11. Scalability of Common Features	203
Analyzing Schema Designs from Gems	204
Understanding Queries From Tagging Gem	205
LIMIT and OFFSET Pagination	207
Database CURSOR Pagination	208
Keyset Pagination	210
12. Working With Bulk Data	213
Creating A Bulk Data Generator Rake Task	214
Batching With Active Record	216
Performing an Upsert in Active Record	217
Handling Conflicts With ON CONFLICT	218
Beyond Active Record With <code>activerecord-import</code>	220
Performing SQL Multi-Row Operations	222
Upserts With SQL MERGE	224
Working With <code>pg_dump</code> And <code>pg_restore</code>	225
Populating Table Data With COPY	226
Creating A File Foreign Data Wrapper (FDW)	230
Wrapping Up	233

13.	Scaling With Replication And Sharding	235
14.	Boosting Performance With Partitioning	237

Part V — Advanced Uses

15.	Advanced Usage and What's Next	241
A1.	Installation Guides	243
A2.	psql Client	245
A3.	Getting Help	247

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B2.0: September 13, 2023

Added new chapters:

- Chapter 11, Scalability of Common Features
- Chapter 12, Working With Bulk Data

Various updates and fixes:

- Minor updates to the text to fix typos

B1.0: August 30, 2023

- Initial beta release.

Preface

If you're looking to improve your skills with PostgreSQL for Ruby on Rails you've come to the right place. In this book you'll work on exercises from practical problems that will help you grow your career.

Maybe your application has grown in popularity and you've got performance problems. Maybe your database has ballooned in size and you aren't sure how to manage the growth. In the job market, while the traditional Database Administrator (DBA) role has declined in popularity¹, there are more database-backed web applications than ever before, and the needs of the role are shifting. Organizations from startups to huge companies choose PostgreSQL as a mission critical database to build their businesses on. Operator challenges, especially under high growth and high query volume, can put a lot of pressure on the engineering team. Maybe you'd like to step up and take these challenges on?

Get ready to develop useful skills for operating PostgreSQL that you can immediately put to use. You'll use the latest versions of PostgreSQL and Ruby on Rails. You'll work on making your applications and databases faster, more reliable, and more resilient. As you work towards mastery of high performance operations, you'll raise the bar for operational excellence for your PostgreSQL database and your engineering team.

As you go through the book's exercises, you'll work with a real Rails application and database schema that you'll evolve as a test lab for your skill development. You'll populate millions of rows of data to work with, to help simulate your production workload. In addition to the core functionality of PostgreSQL and Ruby on Rails, you'll be working with more than 40 Ruby gems and PostgreSQL extensions.

1. <https://builtin.com/software-engineering-perspectives/database-administrator>

Who Is This Book For?

This book is for developers looking to improve their skills with PostgreSQL and Ruby on Rails. Whether you're working on consumer scale Internet applications or enterprise B2B Software as a Service (SaaS), scaling PostgreSQL and Rails application codebases is mission critical for the success of your business. Your team expects to build on and operate these technologies while managing huge data growth, amidst shifting business priorities.

If your job responsibilities or career aspirations include any of the following descriptions, you've come to the right place:

- Ruby on Rails Application Developers improving their PostgreSQL knowledge and skills
- PostgreSQL Database Administrators (DBAs) learning about Ruby on Rails and Active Record
- Infrastructure Engineers developing SQL knowledge and PostgreSQL knowledge including Replication, Maintenance, and Partitioning
- Database Reliability Engineers (DBRE) implementing Sharding, Replication, and Partitioning
- Developers with experience in other relational databases looking to learn PostgreSQL
- Web Application Developers using other full-stack web frameworks like Laravel² or Django,³ looking to learn Ruby on Rails and Active Record
- Data Engineers working with OLAP databases interested in Replication and Change Data Capture (CDC) with PostgreSQL
- Engineers looking to get a promotion to by improving their technical skills with databases

Here you'll work exclusively with the open source community distribution of PostgreSQL. By sticking with PostgreSQL installed locally, readers will gain confidence by developing and testing with a fast feedback loop and without risk.

The focus will be on Transactional workloads, also called Online Transaction Processing or (OLTP), and not Analytical workloads. OLTP workloads are the types of queries you'd see in a web application that is user facing.

2. <https://laravel.com>

3. <https://www.djangoproject.com>

What's Not Covered In This Book?

This book will help you a lot but it doesn't cover everything. It is not an introduction to PostgreSQL or Ruby on Rails. Books like *PostgreSQL: Up and Running*⁴ focus on being an introduction to PostgreSQL. *Agile Web Development with Rails* 7⁵ introduces and covers Ruby on Rails extensively. *Programming Ruby 3.2 (5th Edition)*⁶ is a good choice for developing skills with the Ruby programming language.

Internals of PostgreSQL are not covered beyond some basics. Concepts like Multiversion Concurrency Control (MVCC) and Transaction Isolation Levels are introduced as needed in chapters, but readers are directed to the PostgreSQL documentation for greater depth.

User Administration concepts like, Roles, Grants, and Privileges are not covered. Row Level Security (RLS) and Policies are not covered.

Readers will set up multiple PostgreSQL instances, however High Availability (HA) concepts like replication, Availability Zones (AZ), Regions, failover, and Disaster Recovery (DR) are outside the scope of this book.

For Database Reliability Engineering (DBRE) concepts like Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO), consider the book *Database Reliability Engineering*.⁷

Database Backups are critical. Your organization must collect Snapshots automatically and be able to restore a database server instance from a backup if a disaster strikes. However, backups and restores are also outside the scope of this book. The specific techniques would depend on how PostgreSQL is running in production, which could be from a cloud provider, self-hosted, or containerized.

Finally, this book will focus on the open source community version of PostgreSQL solely. Although there are PostgreSQL forks, PostgreSQL compatible databases, and PostgreSQL extensions significantly modify how PostgreSQL works such as the Citus extension (See: *Citus 11.1 open source release*⁸), these will be outside the scope of this book.

4. <https://www.oreilly.com/library/view/postgresql-up-and/9781491963401/>

5. <https://pragprog.com/titles/rails7/agile-web-development-with-rails-7/>

6. <https://pragprog.com/titles/ruby5/programming-ruby-3-2-5th-edition/>

7. <https://www.oreilly.com/library/view/database-reliability-engineering/9781491925935/>

8. <https://www.postgresql.org/about/news/announcing-citus-111-open-source-release-2511/>

Ruby on Rails Skills Are In Demand

In the piece called “Big Transitions in the Tech Industry” covering Hired’s *2023 State of Software Engineers Survey*,⁹ Ruby on Rails was the most sought-after skill.

Ruby on Rails surfaced as the most in-demand skill for software engineering roles, creating 1.64x more interview requests for the developers proficient in it.

Ruby on Rails is a great way to build PostgreSQL database applications. Ruby on Rails has a Rails Guides page that is dedicated to showing how to use Active Record with PostgreSQL.¹⁰ Some of the capabilities are exclusive to PostgreSQL.

In the earlier days of Ruby on Rails and the Active Record ORM, there was an emphasis on portability of an application codebase to between different relational database systems. Portability has been deemphasized over the years. This is due in part to the longevity of PostgreSQL, which is in turn due to some of the things that have made PostgreSQL successful: standards adherence, features, performance, reliability, permissive licensing, and cloud support to name a few reasons.

Rails continues to add support for PostgreSQL capabilities in each new release. Generated Columns from PostgreSQL 12 were added as Virtual Columns.¹¹ Common Table Expressions (CTE) queries are added to Rails 7.1.

With native support for Multiple Databases,¹² Active Record can use multiple PostgreSQL database instances in a variety of configurations. Multiple Databases can be leveraged for Read and Write splitting and application level sharding either horizontally or vertically. You’ll see how to configure these later in the book.

Skills with Ruby on Rails are in demand. What about PostgreSQL?

PostgreSQL Is A Popular Award Winner

You’ve made a great choice to invest your time learning PostgreSQL.

9. <https://hired.com/state-of-software-engineers/2023/>

10. https://guides.rubyonrails.org/active_record_postgresql.html

11. <https://blog.saeloun.com/2022/01/25/rails-7-postgres-support-for-generated-columns.html>

12. https://guides.rubyonrails.org/active_record_multiple_databases.html

For deployed Ruby on Rails applications, PostgreSQL is the #1 most popular database according to the 2022 Ruby on Rails Community Survey with data from 2009 through 2022.¹³

The *DB-Engines Ranking*¹⁴ ranks the most popular databases in the world. PostgreSQL has been a three-time #1 Winner there in 2017, 2018, and 2020.

In 2021, PostgreSQL was the second most popular database, being a runner-up to Snowflake.¹⁵

In 2022, PostgreSQL was the third most popular database, although effectively the most popular OLTP database, since Snowflake and Google BigQuery¹⁶ were #1 and #2 overall but neither are suitable for OLTP.

The 2022 Stack Overflow Developer Survey¹⁷ gathered input from nearly 50,000 Professional Developers. When the developers were asked what database they used most, PostgreSQL took the #1 spot.

Your investment in yourself building skills with PostgreSQL and Ruby on Rails is a smart career move. These are very popular technologies that are in high demand, used by companies ranging in size from small startups, to giant corporations. Both of these vibrant open source technologies are being improved year after year by a global base of contributors.

The future is very bright for PostgreSQL and Ruby on Rails!

13. <https://rails-hosting.com/2022/#databases>

14. <https://db-engines.com/en/ranking>

15. <https://www.snowflake.com>

16. <https://cloud.google.com/bigquery>

17. <https://survey.stackoverflow.co/2022/#most-popular-technologies-database-prof>

Part I

Getting Started

An App to Get You Started

Welcome! PostgreSQL and Rails are mature, powerful, and modern technologies. If you're here to level up your skills using these technologies together, you've picked up the right book.

The best way to learn is from hands on practice and experimentation. Here you'll work on exercises using practical examples, with a Rails API app connected to PostgreSQL. You may want to challenge yourself on topics you're familiar with to build your own examples. Growth happens when solving challenges just beyond your current skill level.

In this chapter, you'll primarily get things installed and set up. You may have a lot of experience with Ruby on Rails and less or none at all with PostgreSQL. On the PostgreSQL side, you may be an experienced administrator but know little about Ruby on Rails, or perhaps use another popular web framework. Here you'll get information wherever you fall in those skill areas.

The Ruby on Rails app you'll use here is called *Rideshare*, and it's proudly configured to work with PostgreSQL. This is an open source Rails application that's available to download from GitHub.

Rideshare serves as a demonstration application both for Rails and Active Record concepts, and for PostgreSQL concepts. When the topic involves working directly with PostgreSQL, you'll use the Rideshare database. When the topic is a Rails topic, you'll work with the Rideshare Ruby codebase.

What's the Rideshare app all about?

What Is Rideshare?

Rideshare is an API-only web application that implements a portion of a fictional ridesharing service. Think of Uber or Lyft. Imagine there could be mil-

lions of riders booking trips provided by thousands of drivers in cities all around the world.

Although a global ridesharing application would likely have a very large codebase covering dozens or hundreds of models and database tables, Rideshare has less than ten models and tables. The portion of the domain model is a small core or sample that's needed mainly for demonstration purposes in examples and exercises you'll see.

The core nouns you'll work with are *Drivers*, *Riders*, *Trips*, and *Trip Requests*. These are Active Record models that are backed by database tables.

As a refresher or if you're newer to Rails, Active Record is the Object Relational Mapper (ORM) framework for Ruby on Rails. Active Record connects object oriented Ruby classes as application code, to the persistence layer. Active Record Objects are instantiated in memory and when they're saved or updated, a corresponding INSERT or UPDATE is sent to PostgreSQL.

Active Record works with many naming conventions that are important to know. Models use a singular form for names like "Driver" in memory but by convention are persisted into a table with a pluralized name like `drivers`. The *Convention Over Configuration* design principle that's a code aspect of Ruby on Rails means that things like a table name in a model does not need to be configured explicitly.

Rideshare has an *Entity Relationship Diagram* (ERD) available as a PDF in the root directory of the project. The ERD is automatically updated when schema changes are made. See the footnote with a link to the ERD where you can visualize the database tables and their relationships.¹

Active Record Schema Management Refresher

Active Record tracks database schema changes in a file. The file may be in a Ruby format or a SQL format. Rideshare uses the SQL format. After every schema modification is made, the Rideshare database schema is dumped into the file `db/structure.sql`, which is version controlled. This file will always have the latest version of the schema.

When you're working with a new Rails application, explore the `db/structure.sql` file and any available diagrams like ERDs, to begin to learn about the tables, indexes, and constraints.

1. <https://github.com/andyatkinson/rideshare/blob/master/erd.pdf>

If you're newer to Rails, you might be wondering how changes are captured in the `db/structure.sql` file. When a developer makes a schema modification, behind the scenes Ruby on Rails when configured for PostgreSQL uses `pg_dump` to dump the table structure to `db/structure.sql`.

If you've used a schema modification management tool like Flyway² or Liquibase,³ Active Record Migrations are going to be somewhat similar. In Ruby on Rails, Active Record is built-in and is the way developers evolve the schema.

Each modification is called a "Migration", which is an overloaded term. In Rails a Migration is a versioned Ruby file in the `db/migrate` directory. Rails Migrations use an Active Record API that maps Ruby methods to SQL DDL changes.

Within the Ruby source code file, schema modifications can be written in SQL as well. Do Rails Migration files only contain structural changes, aka Data Definition Language (DDL)? No, there aren't any limitations on them, so developers can perform Data Modification Language (DML) statements in the form of inserting, updating, or deleting data. However, it's considered a good practice to use Migrations only for DDL and use Rake tasks or other mechanisms for DML.

Does the `db/structure.sql` contain only structural data about the schema? Although nearly all of the content of the file is purely structural, there is one exception. At the bottom of the file there is "data". The versioned Migration files each have a unique number. The number is inserted into the developer's database when the Migration runs.

How is this tracked then? When the structure file is dumped, it's dumped based on the developer's database, so any migrations that have been applied there are dumped as INSERT statements. The schema migrations are dumped as INSERT statements so that the database structure can be replicated on another database by running `bin/rails db:schema:load`.

What this task does is load the content from the structure file and then insert all the same migration versions into the destination database. That way if another migration comes in after that one with a version number that doesn't exist, the database knows which versions have already been applied and will apply only the new ones.

2. <https://flywaydb.org>

3. <https://www.liquibase.com>

That was a quick refresher on how Active Record Migrations manage the PostgreSQL database structure, how developers make incremental schema modifications, and how those modifications are tracked.

When you clone the Rideshare application and “migrate” (a verb in Rails speak) the database, you’ve evolved your database schema to be in sync with the latest production version.

Now that you have an idea of what the Rails application is, how the database structure evolves and how to get a copy of the structure, you’ll want some data.

You’ll use a couple of ways to get data into your database. To make it easy to get started, you can use a pre-made data dump that can be downloaded and loaded.

Then you’ll learn how to populate even more data via scripts that are part of the Rideshare codebase.

It’s time to get Rideshare set up locally so you can explore the source code and start the database server and web server.

Installing Rideshare

Begin by installing the dependencies needed to run Rideshare. The dependencies are the Ruby language environment, Ruby on Rails framework, Ruby gems, and the PostgreSQL database and extensions.

The examples here were developed on Mac OS using the latest versions of Ruby on Rails and PostgreSQL; these are listed below.

- Mac OS Ventura (2022)
- Ruby on Rails 7 (2021)
- PostgreSQL 15 (2022)

Run the rails executable that’s part of the application by running `bin/rails` from the root directory of Rideshare. You’ll use `bin/rails` to run commands like `console`, `server`, and `runner`. You might see instructions to install Rails using `gem install rails`. This method will make a rails executable available, but don’t use this version for these exercises because it may be the wrong version.

If you’re an experienced Rails application developer and PostgreSQL user, and comfortable with git, you may be able to skip the next few sections and install things as needed.

If you're less experienced or prefer to make sure everything is installed correctly upfront, go through the instructions below to get everything running on your local development computer.

By the end you'll have Rideshare fully operational and connected to your PostgreSQL database server.

Installing PostgreSQL on Mac OS

PostgreSQL can be installed in a variety of ways on Mac OS. The recommended method is PostgresApp <https://postgresapp.com>. Navigate to the website and find the Downloads section, choosing either the Intel or Universal versions, such as “Postgres.app with PostgreSQL 15 (Universal)” for the Universal version. This downloads a .dmg file. Double click the file and then copy Postgres.app into your Applications directory.

Launch the Mac OS application. Once launched, you'll have no database initialized. Choose the plus icon to Create a new server. Take note of the name, version, and data directory. By default PostgreSQL will run on port 5432. Once your server is created, click the Initialize button to initialize it.

You'll get at least 3 databases for your server instance, one named the same as your Mac OS user, one called “postgres” and one called “template1”. Click “Server Settings” to review the file paths to your data directory, log file, or config file. You'll work with those more in upcoming sections.

Follow the instructions below from the PostgresApp site to configure your \$PATH. Run the following commands from your Terminal.

```
sudo mkdir -p /etc/paths.d &&
echo /Applications/Postgres.app/Contents/Versions/latest/bin | sudo tee /etc/paths.d/postgresapp
```

Restart your Terminal. Once restarted, client programs like psql should now be in your system path.

Try checking the versions of client programs like pg_dump and psql⁴ to make sure they're installed.

Open your Terminal and run the following commands.

```
psql --version
pg_dump --version
```

Earlier you saw that you'd have a database created that matches your OS user (type: whoami to find this out). Try running psql without any arguments

4. <https://www.postgresql.org/docs/current/reference-client.html>

from your Terminal. You'll connect to this special database for your user. Type `SELECT CURRENT_DATABASE();` from the `psql` prompt to confirm this. The `SELECT` statement and `CURRENT_DATABASE()` function is not case sensitive, but is shown here capitalized as keywords. Type `\q` to quit `psql`.

Great, your PostgreSQL server is running, and you're able to connect to one of the databases that was created by default. Next you'll get your Ruby environment up and running.

Installing and Configuring Your Ruby Environment

Rideshare is a Ruby application, so you'll need Ruby running on your computer. Ruby includes the `gem` executable, which can be used to install gems. From your Terminal, type `ruby -v` to see what version of Ruby is installed by default.

The system installed version is generally a bit behind. You'll need the specific version that's supported by Rideshare. To find the specific version, Ruby apps by convention place the version in a file called `.ruby-version` at the root directory. Ruby Version Managers were created to help you install and manage multiple versions of Ruby at the same time, so that you could have different versions for different projects. Version managers automatically switch to the correct version when it's installed.

The Ruby version manager `rbenv`⁵ is used here. In this section you'll install `rbenv`.

Once Ruby is installed you'll install the Bundler gem.⁶ Bundler will take care of all the other Ruby gem dependencies needed for Rideshare.

The instructions below are minimal. Extended instructions can be found in the Installation Guides Appendix. Instructions will be maintained at https://github.com/andyatkinson/development_guides as well.

On Mac OS, you'll need to install the Homebrew⁷ package manager, which you'll then use to install the Ruby Version Manager `rbenv`.

Install Homebrew using the latest available instructions at <https://brew.sh>. You'll need to install the Xcode Command Line Tools on new Mac OS machines where it is not already installed. This process starts automatically during the

5. <https://github.com/rbenv/rbenv>

6. <https://bundler.io>

7. <https://brew.sh>

Homebrew installation if needed. Once Homebrew is installed, you'll also need to add it to your system path as you did earlier with PostgresApp.

When Homebrew is installed and added to your path, run the `brew --version` command confirming that version 4 or great is installed.

With Homebrew installed, you're now ready to use it to install the `rbenv` Ruby version manager.

Run the following `brew` command to install `rbenv`. This will also install `ruby-build` automatically.

```
brew install rbenv
```

To verify these are installed, run `rbenv --version` and `ruby-build --version`.

With `rbenv` installed and Ruby versions available from `ruby-build`, you're now ready to install the exact version of Ruby needed for Rideshare. You'll do that in the next section.

Installing the Rideshare Application

You're now ready to install Rideshare. From your Terminal, you'll use `git` to clone the application source code. Run `git --version` to confirm you're running at least version 2. On a recent Mac the version of `git` that is pre-installed may be new enough.

1. Download the application source code to your source code project directory, and change into that directory with `cd`. If you don't yet have a `Projects` directory, type `mkdir ~/Projects` to make one, and then `cd ~/Projects` to change into it. From there, run `git clone https://github.com/andyatkinson/rideshare.git` to clone the source code.
2. Once cloned, change into the Rideshare by typing `cd rideshare`

If you type the `pwd` command, you should now be located in a directory like `/Users/andy/Projects/rideshare`.

3. From there, you're ready to install Ruby and the Rideshare gems.

Use `rbenv` that you installed earlier, to install the version of Ruby listed in `.ruby-version`. For example, run `rbenv install 3.2.2`. This will take a minute as it compiles Ruby for your system. Once that's installed, run `ruby --version` again and confirm the version matches. Now you're ready to install Bundler. Install the bundler gem by typing `gem install bundler`. This is the only gem you'll install using the `gem` executable. All other gems are

installed using Bundler. Type `bundler --version` to confirm that you have version 2 or greater installed.

With bundler installed, you're ready to install the Rideshare gems. From the Rideshare directory run `bundle install`. This command installs the Rails gems and all the third party gems needed to run the application.

If you run into errors, please review the Installation Guides Appendix or open an issue on the Rideshare project.

4. Once all the gems are installed successfully, you're ready to set up the database. You'll now run `bin/rails` commands provided by the Rails framework. From your Terminal:

Run `bin/rails db:create` to create the development and test databases. The PostgreSQL server must be running. The database will be owned by `postgres` by default.

5. You should now be able to connect to `rideshare_development` by running `bin/rails dbconsole`.

Installation and initial configuration is now complete. To make sure the database schema is updated, run `bin/rails db:migrate` to apply any pending migrations. By default an Entity Relationship Diagram is generated when migrating. You'll need `graphviz`⁸ to keep that updated. Run `brew install graphviz` to get that installed.

You may also wish to run the test suite for Rideshare which you can do by running `bin/rails test`. If everything passes, you're in good shape!

You're now ready to start diving under the hood.

Working With PostgreSQL Locally

In the exercises here you'll work exclusively using your local version of PostgreSQL. Your local PostgreSQL database will have less functionality compared with modern cloud hosted PostgreSQL instances, but it will be less complex and you'll have full control over it. You'll use your local instance as a training ground where you'll develop skills you can take to production.

You'll view log files, modify parameters, and start and stop your server. Since you're working locally on PostgreSQL, any queries or processes are in your control and you may restart it at will. In production a database restart requires

8. <https://graphviz.org>

careful planning and is generally avoided. Thus, you'll practice "reloading" configuration changes where possible to avoid restarts.

PostgreSQL is an extensible database. Here you'll be limited to open source PostgreSQL only, but you'll add extensions from the PostgreSQL ecosystem. Extensions expand the core functionality. The extensions featured here have broad support from major cloud providers of PostgreSQL, so your knowledge and skills will be portable. If you're self-hosting PostgreSQL, you can install and configure any extensions that you'd like.

With that in mind, let's jump into some PostgreSQL terminology.

Learning PostgreSQL Terminology

PostgreSQL is an object relational database. If you're a programmer and are familiar with Object Oriented Programming (OOP) then you might think of objects like Ruby classes and inheritance hierarchies.

In PostgreSQL, database objects are things like tables, indexes, schemas, column defaults, or constraints.

Databases have a lot of terminology you'll want to familiarize yourself with. Early and frequent exposure to terms is important for learning, so there are terminology guides in most chapters that provide short simplified definitions.

Some terms you'll run across are *relations*, *tuples*, *bloat*, and the *optimizer*.

A *relation* refers to a table. *Tuples* are versions of rows in tables. *Bloat* has to do with inefficient data layout, where data is stored in a fragmented way and this fragmentation can add latency to operations. These very brief definitions are a starting point for exposure to these terms.

Some basic information on PostgreSQL data storage internals is useful. There's a lot more that we won't discuss.

In PostgreSQL, data is laid out in *pages*. Data cannot be split across pages. Data is stored efficiently when there are minimal gaps and data that is accessed resides within a single page. Data that's in a single page requires fewer disk operations compared with accessing data that is split across multiple pages.

Another concept on data layout is *correlation*. Correlation refers to the relationship between how data resides on disk and how the data is ordered in a query. High correlation means they're similar and low correlation means they are dissimilar.

Data may not fit in a single page, may be split across pages, or have low correlation. PostgreSQL works whether data is stored efficiently or not.

Next let's look at some SQL terminology.

Learning SQL Terminology

SQL has loads of terms as well, and much of it is common to all relational databases.

SQL is a *declarative* language. This means that you *declare* to PostgreSQL the results you'd like, and PostgreSQL figures out how to get them.

You specify the columns from the tables you want, and PostgreSQL determines an optimized retrieval algorithm using knowledge of the data layout, database objects, and available resources.

PostgreSQL creates *query execution plans* on the fly to fulfill what you've asked for. Multiple plans are pitted against one another. The plans are formed using estimates about data in tables, so the plans aren't always flawless but are generally pretty good. As the programmer, you can see what PostgreSQL is planning and influence it. You'll learn more about that later on.

This plan evaluation process is conducted nearly instantly by the query execution planner, which is also referred to as the *optimizer*.

The optimizer uses a *cost-based algorithm* to compare costs and select the lowest-cost plan. Plan selection is a very interesting and deep topic that you'll also get some experience with.

Besides writing SQL, you'll run statements and commands for PostgreSQL, sometimes using built-in *functions*. Functions enhance your database; there are built-in functions and you may create your own as *User Defined Functions*. Functions can be written using SQL or other programming languages.

Statements you write in PostgreSQL can be classified in a couple of ways. A statement that changes the structure of the database, like creating a table, is called a Data Definition Language (DDL) statement. An example would be a CREATE TABLE statement where you'd supply a table name and the column definitions you'd like.

Another type of SQL statement is one that modifies data and not structure. This statement type is called a Data Manipulation Language (DML) statement. An example is an INSERT statement that inserts data into a table. Database users might use these acronyms DDL and DML in conversation to categorize statement types.

PostgreSQL is a transactional database, which means most operations take place inside of a *Transaction*. Transactions are used to group together multiple operations that should occur as one unit. Transactions have an isolated view of the stored data and the transaction isolation level can even be changed.

Let's recap some of the terms you've seen so far.

Glossary of Database Terms



- Relation or rename — a table
 - Data Manipulation Language (DML) — e.g. INSERT, UPDATE, and DELETE statements
 - Data Definition Language (DDL) — Structure modifications like ALTER TABLE or CREATE INDEX statements
 - Transaction — A unit of work that occurs together
-

Next you'll dive into some Ruby on Rails terminology.

Ruby on Rails Terminology

Active Record is the Object Relational Mapper (ORM) for Ruby on Rails. Active Record classes are Ruby classes. A class also serves as an interface that connects rows in a database table to Ruby object model instances running in memory.

Active Record goes beyond simple persistence and includes lots of data modeling and data validation features. Active Record objects have *Associations* to link models together and *Validations* for validating the correctness of data.

Active Record Terminology



- Object Relational Mapper (ORM) — Maps relational database tables to Active Record model classes in Ruby on Rails
 - Migrations — Migrations are used to make incremental modifications to the database structure.
 - Associations — Configuration in Active Record models like `has_many` or `belongs_to` that describe how models are related to one another
 - Validations — Application level data consistency checks
-

Now that some of the standard PostgreSQL, SQL and Rails terms are clear, let's look at conventions used in the coming chapters.

Conventions Used In This Book

Externalized code snippets are used for shell scripts, SQL statements, and Ruby scripts, to format the code and make it easier to work with outside the book. Code snippets are available for download.

Some choices are intended to make new concepts more memorable. For CLI programs with short and long options like `-v` or `--version`, the longer form will be used.

For meta commands in `psql` like `\du`, they'll be described like “describe user” to better connect the words and short form of the command.

Shell scripts may be run in your Terminal. On Mac OS, use the built-in Terminal application or a third party option like `iTerm`.⁹

To run the `load_data.sh` shell script, from your terminal type `sh load_data.sh`.

SQL Conventions

SQL keywords are capitalized and monospaced like `SELECT`, `WHERE`, and `LIMIT`, to better set distinguish them from table names and column names. Portions of a query that are incomplete are referred as *clauses*.

For example, in the SQL query `SELECT * FROM trips WHERE id = 1`; the portion `WHERE id = 1` is called the *where clause*.

SQL queries are executed from a SQL client. This book recommends the `psql` client.

Once you're connected to PostgreSQL, copy and paste or type the SQL query into your client to run it.

Database functions will be capitalized and have parentheses as well, like `SELECT AVG()`.

Rails Conventions

Rails code is written in the Ruby programming language. Ruby is a interpreted language so you can run any Ruby code without compiling it, by simply copying and pasting it into the “Rails Console”. Launch the Rails Console by running `bin/rails console` from the Rails application root. Run any Ruby code in the console. Running code in `bin/rails console` is a great way to explore Active Record code, model code, or other Rails application code to see how it works.

9. <https://iterm2.com>

Besides the psql client, and the Rails console, SQL statements and Ruby code can be also be run from the Terminal directly.

For SQL, psql supports a `--command` option. For example, type `psql --command 'SELECT 1'` to run a SQL query.

Or try `ruby -e "puts 'hello'"` to run Ruby code from the Terminal within the double quotes. Rails extends this with `bin/rails runner` which allows you to run Rails code from the command line. You'll use the Rails Runner later in the book.

Ruby methods will start with a dot and be in lowercase, with explicit parentheses on the end like `.average()`.

Ruby on Rails Frameworks like “Active Record” or “Action Pack” with multiple words will have their words separated by a space.

You're Ready

You're now ready to jump into some real work.

One of the main goals is to help you improve your PostgreSQL operational and administration skills. Many Rails developers do not typically work on operation and administration directly with PostgreSQL.

To improve your skills you'll need to get comfortable performing tasks, modifying configuration files, viewing logs, and understanding how to observe what's currently happening inside the database.

It's time to start building those skills!

Part II

Design and Build

Administration Basics

Welcome! In this chapter you'll get hands on with PostgreSQL operation and administration.

In the last chapter, you installed and configured the Rideshare Rails application with a local PostgreSQL database. If you skipped the Introduction and aren't familiar with Rideshare or don't have it installed, consider going back and performing that installation and configuration now.

Once that's complete, you'll have a full application development environment installed locally with the correct versions of Ruby, Rails, and PostgreSQL running. You'll be ready to run Active Record code from `bin/rails` console for Rideshare or run SQL statements in the database.

PostgreSQL Administration is a broad subject and in this chapter you'll be getting an introduction to a variety of topics to help you branch out and go into greater depth.

In this book there may be a lot of new terminology to learn. You'll work with meta-commands, `psql`, observability concepts, locks, database parameters, and extensions.

Let's dive in to some basics of administration by working with the Rideshare database.

Learning Meta Commands In `psql` With Rideshare

When connecting to PostgreSQL you'll need a client application. GUI applications are great, but in this book you'll work solely from your Terminal using the `psql` client.

`psql` is recommended for several reasons. Primarily it ships with PostgreSQL, so it's always available. While there might be a steeper learning curve com-

pared with a GUI tool, the immediacy and low latency as a command line client are big benefits. To learn about more of the rationale for recommending psql, view an expanded answer in the Appendix “Why psql?” at the end of the book.

To get started, type psql with the --dbname option specifying the Rideshare database:

```
sh/psql_dbname.sh
psql --dbname rideshare_development
```

This command assumes the host and credentials aren’t needed to connect. If the command doesn’t work as-is, review the “Development Guides” Appendix for more information. Make sure you’re able to easily connect to your database since you’ll be doing that a lot.

You’re now staring at a blank prompt. Now what?

Let’s explore. A backend process ID is running that corresponds to the psql session you just started. Run `SELECT PG_BACKEND_PID();` to see the value. You’ll use this information later.

Try typing `\s` and hitting Enter. This is a *meta-command* and it might be the first one you’ve used if you’re new to psql. psql has loads of meta commands. Some are listed in the post *17 Practical psql Commands*¹ and you’ll see a lot of them here. This meta-command lists the history of SQL statements or commands you’ve run in psql before. If you run a statement like `SELECT 1;` and then run `\s` again, you’ll see the statement in the history.

You’ll begin working with extensions and System Catalogs.² From psql, run `SELECT * FROM pg_extension;` to see installed extensions. `pg_extension` is a System Catalog and PostgreSQL has loads of System Catalogs to track all kinds of information internally. System Catalog tables start with `pg_*` and shouldn’t be modified.

Administration Terminology



- Meta Commands — psql commands that start with a backslash
 - Backends — A backend process with a pid and query
 - System Catalog — A PostgreSQL table or view you might use to inspect internal state information
-

1. <https://www.postgresqltutorial.com/postgresql-administration/psql-commands/>

2. <https://www.postgresql.org/docs/current/catalogs-overview.html>

In the same way the history command works in your Terminal, you can do a “reverse search” through your SQL statements by typing “ctrl-r”. After typing ctrl-r, type SEL to find the SELECT statement you just ran. Type \g to run the last statement again.

The history is kept in the file ~/.psql_history by default and this can be customized by setting the HISTFILE option. The history file is a plain text file that’s meant to be read within psql.³

You can increase HISTSIZE to keep more information in the file. Review the HISTFILE and HISTSIZE options on the psql documentation.⁴

Customizations are added to the ~/.psqlrc file that’s read when psql starts up. The Thoughtbot post *An Explained psqlrc*⁵ shows how to customize ~/.psqlrc with lots of great tips.

You may also want to edit queries directly in a text editor from psql. To do that, use the \e meta-command.

Within the text editor, type out SQL queries, wrapping lines and doing formatting as you like. When you save your changes you’re returned to psql and the query executes.

To save your SQL query output to a file from psql use the \o meta command. For example, type \o file.txt to send query results into a file called file.txt.

Output will continue to go into the file until you toggle it back. Toggle it back by typing \o without a filename. You should now see query results again in psql.

To exit psql, type \q. View the query result contents of the output file you just created by running cat file.txt from your Terminal.

Refer to the meta commands recap below.

Command	Description
\s	See history of commands
\e	Launch a text editor from psql
\o	Toggle output between psql and a file query output to a file from

Great, hopefully you’re picking up psql tips and getting more comfortable using it.

3. <https://stackoverflow.com/a/31649807>

4. <https://www.postgresql.org/docs/current/app-psql.html>

5. <https://thoughtbot.com/blog/an-explained-psqlrc>

Next, you'll begin to modify your PostgreSQL configuration.

Modifying Your PostgreSQL Config File

Modifying your local PostgreSQL configuration is a way to practice changes that you might want to perform in your production PostgreSQL instance. With cloud providers, you may use a web UI or CLI program to modify parameters and not edit a config file directly, but the parameter names, values, and change process is about the same. Depending on which parameters you are changing, you'll be able to either reload the configuration without restarting PostgreSQL, or you'll need to restart PostgreSQL for the changes to be applied.

First you'll need to find the config file. From `psql`, run the following statement to find the path:

```
sql/show_config_file.sql
SHOW config_file;
```

On Mac OS, the path includes a directory ("Application Support") with a space in it, so you may need to add double quotes to the path.

Open the file with your text editor and take a look at the content.

```
sh/edit_postgresql.conf.sh
vim "/Users/andy/Library/\
  Application Support/Postgres/\
  var-15/postgresql.conf"

# -----
# PostgreSQL configuration file
# -----
#
# This file consists of lines of the form:
#
#   name = value
#
# (The "=" is optional.)  Whitespace may be used.
# Comments are introduced with "#" anywhere on a line.
# The complete list of parameter names and allowed
# values can be found in the PostgreSQL documentation.
#
# (continued...)
```

Within the `postgresql.conf` file you'll see lots of parameters that can be modified. It's a long file and a snippet from the first few lines is shown below. Search within the file for the section called `shared_preload_libraries`. Most parameters will be commented out.

The default value might be an assignment to an empty string:

```
shared_preload_libraries = ''
```

Next you'll set a value for `shared_preload_libraries` by setting a value inside the single quotes. Before going too far, you may wish to make a backup copy of this file or even check it in to version control. If something goes wrong, you'll want to roll back to a known good version and restart PostgreSQL.

You'll enable an extension that's included with PostgreSQL by configuring it in this section. This extension will be used later, but for now you're just looking to enable it.

Add the `pg_stat_statements`⁶ extension. This extension is used to collect statistics about queries that are executing on the server. It will be abbreviated as PGSS. This is a good extension to make available in general with very useful functionality.

Type `pg_stat_statements` inside the single quotes in your editor like below:

```
sh/shared_preload_libraries.sh
shared_preload_libraries = 'pg_stat_statements'
```

Changing `shared_preload_libraries` does require restarting PostgreSQL to take effect. In your production database server, a restart needs to be carefully planned. Other customizations require only a “reload” and not a restart which is much better because it doesn't interrupt any running database queries or activity.

Since you're working locally and are the sole user of the database, restarting is no big deal. If you're using PostgresApp on Mac OS, you can restart PostgreSQL using the GUI. Click “Stop” and “Start” from the GUI app.

To restart PostgreSQL from your terminal, use the `pg_ctl` program and restart argument. This command expects PGDATA to be set either as an environment variable, or as a command argument. Try restarting it this way and setting PGDATA as an argument. First you'll need to get the data directory for your installation to pass as the value. Let's take a brief detour to do that.

To get the path, run the following command from `psql`:

```
SHOW data_directory;
```

This might be something like `/Users/andy/Library/Application Support/Postgres/var-15`.

Type `\q` to exit `psql` and return to your terminal.

You can now run `pg_ctl restart` and use the `--pgdata/-D` option and your data directory path.

6. <https://www.postgresql.org/docs/current/pgstatstatements.html>

The full command to restart PostgreSQL is below:

```
pg_ctl restart --pgdata "/Users/andy/Library/Application Support/Postgres/var-15"
```

Once PostgreSQL is restarted, run `psql` again. You can now enable the extension per database. Connect to the Rideshare development database and run the following statement.

```
CREATE EXTENSION pg_stat_statements;
```

List the enabled extensions again for the database, and you should now see PGSS.

```
sql/select_pg_extension.sql
```

```
SELECT * FROM pg_extension;
```

Great. You’ve now completed some basics on configuring and enabling an extension. Next you’ll start to explore what’s currently happening in your database while it’s running.

Getting Started With Query Observability

By looking at the schema, tables, relationships and structure, you can get a general sense of what might be happening in the database, but not what’s currently happening. How can you do that?

PostgreSQL makes current activity available from a System Catalog called `pg_stat_activity`.

To view current database activity, launch `psql` and run this SQL query:

```
sql/pg_stat_activity.sql
```

```
SELECT * FROM pg_stat_activity;
```

Try toggling `\x` (another meta-command) to make the output more readable. Even if no application queries are running, you might see things like “auto-vacuum launcher” or “logical replication launcher”, which are background processes running within PostgreSQL.

To inspect details about the `psql` session you’ve opened, you can query `pg_stat_activity` using the `pid` for your current backend process. Let’s do that now. This uses the query you saw earlier to find your backend `pid`.

```
sql/select_pg_backend_pid.sql
```

```
SELECT * FROM pg_stat_activity
WHERE pid = (SELECT * FROM PG_BACKEND_PID());
```

Running this statement in your production database and viewing current activity is safe to do, although it may produce an overwhelming amount of information depending on how many processes are running.

If you see `INSUFFICIENT PRIVILEGES` when running this query, you'll need to connect to your database using an administrator role.

Take a look at the `pid` Process ID field.⁷ If you have a query that's running too long and you want to stop it, you'll need the `pid` value and can cancel or terminate the backend process associated with the query.

These capabilities are part of the PostgreSQL System Administration functions.⁸ The example below shows the format where you'd replace `pid` with the `pid` value.

```
SELECT PG_CANCEL_BACKEND(pid);
```

This form tries to cancel the query, sending a `SIGINT` signal which is a more graceful shutdown attempt.

If that doesn't work, the next option is to terminate the query by sending a `SIGTERM` signal. Terminating a running query may cause data loss.

Multiple queries in progress may be associated with the backend process so only use this option when it's really needed. Use the following form to terminate the backend, replacing `pid` below with the real value.

```
SELECT PG_TERMINATE_BACKEND(pid);
```

Later you'll learn about safeguards you can put in place to help reduce the likelihood of needing to manually cancel or terminate a query.

Another type of query error you'll want to be aware of is queries that are in a blocked state. One of the key design features of PostgreSQL is how it locks and unlocks resources to handle a lot of concurrent writing and reading activity. The trade-off here is that a query can become blocked while waiting for a lock.

In the next section you'll peek into some of the details on lock activity.

Glancing At Current Lock Behavior

PostgreSQL uses pessimistic locking, which means lockable resources like tables or rows are locked up front before they're modified. When queries try

7. <https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW>

8. <https://www.postgresql.org/docs/current/functions-admin.html>

to access lockable resources, depending on the query type it will require either a shared lock or an exclusive lock.

Although a deep dive into locks is beyond the scope of this chapter, knowing some basic information about locks can help you design solutions that minimize lock problems.

Exclusive locks should be minimized because they block other queries trying to access the same resources. You'll learn more about how to do that, and how to add safeguards to avoid waiting for too long.

Another concept to be aware of is that the locks are in effect for some period of time, and the period of time is relevant to determining the impact. For locked tables with high row counts, locks will be in effect for longer periods. The correlation between lock length and row count is there because every existing row may need to change for certain table structural changes. If the same table that's changing is being queried, these queries can be blocked, cancelled, and those cancellations will result in application errors.

Although Locks can be created explicitly, usually resources are locked “implicitly” based on the lock type the SQL statement requires. You will learn to map the SQL statements you send to the lock types they require in future chapters. If you want to get a jump start on that, Hussein Nasar did the heavy lifting already creating the *PostgreSQL Lock Conflicts*⁹ tool which shows all the lock types, and even maps specific SQL commands to the lock types they take. Besides that, the PostgreSQL documentation shows which lock modes conflict which each other on the *Explicit Locking Documentation* page.¹⁰

A type of lock problem to be aware of are *Deadlocks*. Deadlocks happen when two processes block each other permanently.¹¹

For live information about locking activity, query the `pg_locks` system catalog from `psql` as follows.

```
sql/view_pg_locks.sql
SELECT * FROM pg_locks;
```

That concludes a brief introduction to lock activity. To recap, `pg_stat_activity` and `pg_locks` are system catalogs that can be queried to view live activity or lock activity.

9. <https://postgres-locks.husseinnasser.com>

10. <https://www.postgresql.org/docs/current/explicit-locking.html>

11. <https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-DEADLOCKS>

Sometimes you'll want to simulate behavior that you wish to monitor, like slow queries or long running locks. Since those lock durations can be very short, you'll want to create some experiments to explore this behavior more directly.

Read on to learn how to create these types of experiments.

Generating Fake Data For Experiments

So far you've created a database for the Rideshare application and connected to it. Your database is running on your local computer, which is the server or "instance" that hosts the database.

Did you know your database server instance has many databases running on it? To clarify the "database" term, we'll take a brief detour. In a default PostgreSQL installation, you'll have databases called `postgres`, `template0`, and `template1`. You've also created the database `rideshare_development` and `rideshare_test`. These are all on the same server instance.

Let's explore these databases. Run `psql` and type the `\l` meta-command to *list* the databases.

From there, type `\c postgres` to *connect* to the `postgres` database.

A database is a logical grouping of tables and other objects that exists on your PostgreSQL server instance.

Let's make another database that's dedicated to experimentation, so that it's isolated from the Rideshare database. This will provide another place to experiment with schema modifications, bulk loading, or query performance.

From `psql`, type `CREATE DATABASE experiments;` to create the new database. Connect to it with `\c experiments`.

Next you'll create a table in your `experiments` database and populate it with fake data.

Try writing a `CREATE TABLE` DDL statement in SQL to create a table called `tbl` with a single column `col` with type `smallint`.

Normally you might use Active Record to create a table. It's ok to look up the syntax for this. The goal is to get familiar with performing these types of operations using SQL.

Next, insert some data into the table you've made. How might you insert 10 rows into your new table? You'll have an easier time if you use the built-in `GENERATE_SERIES()` function. This is a function available in PostgreSQL to generate

a series of values that could be Integers, Dates, or something else. Use this function to generate values for 10 rows.

The SQL statements below create the table and populate the rows:

```
sql/create_table_generate_series.sql
-- create the table
CREATE TABLE tbl (col SMALLINT);

-- populate the table
INSERT INTO tbl(col) SELECT GENERATE_SERIES(1, 10);
```

You've now created a new experiments database, a table, and populated it with rows.

What else could you practice as SQL?

Creating Indexes Using SQL

As a Ruby on Rails developer, you may not have written many CREATE INDEX statements in SQL. You've used the Active Record `create_index` method which in turn generates the SQL statement for you.

Try creating an index using SQL. In query performance emergencies or for certain types of exploration, being able to create an index from psql can be a useful technique.

Deploying an Active Record Migration may take a lot of time depending on how fast your code review and software release process is, or on how many database changes the DDL statement needs to run.

If you are very uncomfortable with the idea of direct database modifications, that's ok, your intuition is right. In general you want database structure changes to be managed with a change management process and reviewed by team members. However, here you're looking to build your SQL DDL skills and you're only working on your local instance.

From psql, create an index called `test_index` for the `tbl` table in the experiments database you created earlier.

This index should cover the `col` column on the `tbl` table and be the default B-Tree type. Don't worry about the purpose of the index; right now you're just getting comfortable developing and running the statements.

Type `\timing` to toggle on timing.

With timing enabled you'll see how long the CREATE INDEX statement takes. On large tables it can take a long time to create an index.

Adding indexes on live systems in production should be done using the `CONCURRENTLY` keyword, which means the table can continue to be queried while the index is being added. The trade-off with `CONCURRENTLY` is that it makes the operation take about twice as long. Feel free to skip it on your local instance.

The SQL statement is below:

```
sql/create_index_basic.sql
```

```
CREATE INDEX test_index ON tbl (col);
```

Try dropping the index you just created using `DROP INDEX`. From `psql`, run `DROP INDEX test_index;`. Note that indexes should also be dropped concurrently on production systems.

When you've made a structural change you'd like to undo, how can you do that?

Rolling Back Schema Modifications

PostgreSQL has a feature called *Transactional DDL*.¹² Transactional DDL means that when structure changes are made, they either fully succeed or are fully rolled back, avoiding a partially applied modification.

Speaking of transactions, what is a *database transaction*? Rails developers may be less familiar with database transactions. Here are the basics.

Transactions help facilitate making multiple modifications to structure or data as one unit. Transactions can be explicitly created, but are usually implicit from the statements you're running.

Try out Transactional DDL a bit with an experiment. Try making a DDL change and use an explicit transaction that you create. You'll roll it back and confirm the change is not applied.

Open up `psql` and type `BEGIN;` to open a transaction.

Inside the transaction create a DDL change like adding the index using the statement you just created in the last section.

Since you're using an explicit transaction you have "transaction control." With transaction control you may decide to roll back the transaction. After typing the `CREATE INDEX` statement and a semi-colon to end the line, on a new line type `ROLLBACK;` as follows.

```
sql/transactional_ddl_changes.sql
```

```
BEGIN;
```

12. https://wiki.postgresql.org/wiki/Transactional_DDL_in_PostgreSQL:_A_Competitive_Analysis

```
CREATE INDEX test_index ON tbl (col);
ROLLBACK;
```

The ROLLBACK keyword rolls back a transaction, and thus the Index should not be created.

Now that you've seen Transactional DDL, it may be tempting to experiment with structural modifications in your production database. You can roll them back. But is this a good idea?

Exploring and Experimenting Safely In Production

Most companies have a staging environment that's running the application with a PostgreSQL database separate from production.

A staging environment is a great place to perform some kinds of experimentation. Take advantage of this if it's available to you! This is a much better place to perform structural modifications or load large amounts of data without risk to production.

Consider creating a read only PostgreSQL user on the primary database using the `pg_read_all_data` or `pg_monitor` role. The article *Creating a Read-Only Postgres User*¹³ from Crunchy Data describes how to do this. By regularly using a read only user, you'll add another safety mechanism that prevents accidental modifications.

New tables, new constraints, new data relationships, and new indexes are common ways that applications evolve as developers add more features or improve the data model over time. While schema evolution happens locally during the development cycle, it's not always practical to populate huge amounts of data in local databases to do larger scale performance testing. Larger scale performance testing may in turn reveal opportunities for structural changes like adding Indexes to a table.

Wouldn't it be nice to have a performance database environment to test these kinds of changes on? How might you set that up?

In the next chapter you'll set up a Performance testing database where you can make these kinds of changes. You'll build up a Performance database by starting from the Rideshare database, making the data safe to work with, and learn some techniques to make the process scalable and repeatable.

Let's get started!

13. <https://www.crunchydata.com/blog/creating-a-read-only-postgres-user>

Building a Performance Testing Database

In the previous chapter you started working directly with PostgreSQL from `psql` and learned about the Rideshare application. You ran meta commands, inspected system catalogs, and configured an extension.

Moving along, you're about to learn to develop your own copy of your production database where you can begin to experiment with more extensive types of changes including structure modifications. You can also use this database for Performance testing, so it will be called the Performance database.

You'll begin to use advanced PostgreSQL concepts like *Functions* and *Procedures* by solving the very common problem faced by development teams, which is how they can get production-like data in their pre-production environment, without compromising any sensitive customer data. Some of the reasons to have production-like data are for performance testing of queries or for experimenting with indexes or different data model designs.

You'll write custom functions using SQL and learn about the *PL/pgSQL* Procedural Language. If you've never worked with this before don't worry, no prior knowledge is expected.

As your application scales up to higher amounts of queries and tables with greater row counts, having a place to test query changes and structure modifications becomes even more valuable.

You'll work with a pre-made database dump file for Rideshare that contains data that simulates sensitive customer information. Rideshare is fictional and the generated users are fictional as well, but you can treat this data as if it were real data like you'd find in your production database.

When columns have values that uniquely identify individuals they're considered to be sensitive. A good security practice is to limit access to sensitive

data. To do this, you'll build a *scrubbing* process that replaces sensitive values with non-sensitive values.

A design goal of the scrubbing techniques you'll use is that they'll work quickly even for tables with millions of rows. This provides an opportunity to discuss some of the ways that row modifications can slow down in PostgreSQL.

Another design goal is that the *scrubbed* data maintains the statistical properties of the original data as much as possible. This way your query performance analysis in your Performance database will very closely match your production database.

The scrubbing process works with all modern versions of PostgreSQL and has no external dependencies. To achieve that level of flexibility, this means you'll be building all of the pieces yourself!

To get started, first you'll download a copy of a pre-made data dump.

Downloading Production Data Dumps

The Riders and Drivers in Rideshare were generated using fake values from the Faker gem¹ which produces values designed to look real.

A pre-made dump of one million rows was made using `pg_dump` as below. You don't need to run this now, it's just here to show you what was done earlier.

```
sh/dump_rideshare_database.sh
# create the dump file
pg_dump --dbname rideshare_development \
  --no-owner \
  --no-privileges \
  --file tmp/rideshare_database_dump.sql

# compress the file, force overwrite
gzip -f tmp/rideshare_database_dump.sql
```

Once the dump file was created, it was uploaded to GitHub. You'll download it from there.

The amount of records was chosen to create a dump file that was large enough to be “interesting”, but small enough to download quickly. For performance testing it helps to have large amounts of row data to work with. After downloading the dump you may wish to generate even more data, which you'll see how to do.

1. <https://github.com/faker-ruby/faker>

Dumping the Database Schema

You may have been using `pg_dump` with Ruby on Rails and not realized it. As the database schema is modified, changes show up as diffs to one of two possible files, either `db/schema.rb` when the Ruby format is used, or `db/structure.sql` when the SQL format is used. Rideshare uses the SQL format. The SQL format uses `pg_dump` behind the scenes.

You're now ready to download the dump file and load it. From the Rideshare directory, run `sh reset_and_load_data_dump.sh` in your Terminal. The commands are also listed below.

```
sh/reset_and_load_data_dump.sh
#!/bin/bash

# Important Note:
# Run this script from parent directory
#
# cd ..
# sh scripts/reset_and_load_data_dump.sh

echo "Make tmp dir"
mkdir -p tmp

echo "Download dump file"
curl -L \
https://github.com/andyatkinson/rideshare\
/raw/master/rideshare_database_dump.sql.gz \
  -o tmp/rideshare_database_dump.sql.gz

echo "Decompress file"
gzip -d tmp/rideshare_database_dump.sql.gz

echo "Decompressed file is around 100 MB"
du -h tmp/rideshare_database_dump.sql

echo "Re-create the empty database"
bin/rails db:drop:all
bin/rails db:create

echo "Load the dump file using psql. \
This will take a minute..."
psql --set ON_ERROR_STOP=on --quiet \
  --no-psqlrc \
  --output /dev/null \
  rideshare_development \
  --file tmp/rideshare_database_dump.sql

echo
echo "done!"
```

After the script runs, the `rideshare_development` database should be populated with one million records for the users table. The Rideshare users table is a polymorphic table and stores Drivers and Riders.

Run `psql --dbname rideshare_development` and count the users table records to verify that everything was loaded.

```
SELECT COUNT(*) FROM users;
```

Describe the users table by running `\d users` from `psql`. Notice users has fields like `first_name` and `email`. The values in these fields are unique to individuals are sensitive. These are some of the fields you'll scrub!

How Do I Identify Sensitive Data?

To identify sensitive data, you'll look through all of the columns and find ones that describe personal information.

With the downloaded and replaced `rideshare_development` database, you've got lots of data and it has not yet been scrubbed. This will be your starting point from where you'll run your scrubbing process. If you wish to start over at any point, recreate the database (`bin/rails db:reset`) and run the script again.

You're now ready to jump into the scrubbing process itself. The first design goal to tackle is a desire to keep the statistical properties the same after scrubbing data. What does that mean?

Replacement Values That Are Statistically Similar

The statistical properties of the data in tables, columns, and indexes all affect query execution performance.

Query execution plans are determined dynamically at execution time. PostgreSQL uses estimates and statistics about the data to estimate the cost of alternative plans. The query planner chooses the lowest cost plan that produces the fastest data retrieval. We'll get into the details of query execution plan generation and plan selection later; for now, it's enough to know that table statistics are important to query planning.

What are some of the statistics collected by PostgreSQL?

To find out, try exploring the `pg_stats` System Catalog view.² The statistics collected here include things like the most common values or how many rows have NULL values in their columns.

For the Rideshare database, take a look at the statistics captured for the `users` table by running the following query from `psql`. An example result row is shown below.

```
sql/table_statistics_users.sql
```

```
SELECT
    attname,
    n_distinct,
    most_common_vals
FROM pg_stats WHERE tablename = 'users'
AND attname = 'first_name';

-- # SELECT attname, n_distinct, most_common_vals FROM
-- pg_stats WHERE tablename = 'users' AND attname = 'first_name';
--
-- -[ RECORD 1 ]-
-- attname          | first_name
-- n_distinct        | 5122
-- most_common_vals | {Elroy,Maurice,Tristan,Dion,Ariel,Angel,Chi,
-- Chris,Frances,Luis,Mark,Ray,Fermin,Guadalupe,Logan,Perry,Royce,
-- Scott,Vernon,Carey,Carson,Cary,Dalton,Erin,Eugene,Frank,Glenn...
```

The commented out portion shows an example record.

[RECORD 1] shows that 5122 distinct first names are captured in the `n_distinct` column.

You can see some of the most common first names like Elroy, Maurice, and Tristan in the `most_common_vals` column. “Elroy” is one of the most common first names. The `most_common_freqs` column shows the frequency at which the most common values occur.

Calculate the frequency of Elroy as a first name. The frequency is the total number of occurrences of “Elroy” divided by the total number of rows in the `users` table. Since Elroy appears 405 times, divided by 1,000,000 total rows, it has a frequency of 0.000405.

The *cardinality* of a column refers to its distinct values.

2. <https://www.postgresql.org/docs/current/view-pg-stats.html>

Performance Database Terminology



- Schema — In PostgreSQL it's a namespace
 - Scrubbing — Replacing sensitive text in your database fields
 - Functions — Database functions in SQL or the built-in procedural language PL/pgSQL
 - Procedures — An extended form of functions with more capabilities
 - Cardinality — Unique values in a column
-

For query performance analysis it's ideal to closely match replacement text with the original text to keep characteristics like cardinality and frequency similar to what's in production. Besides that, the space consumed within the column is worth keeping as close as possible.

"Elroy" is a string of five characters. "Elroy" is stored in a character varying column. A good replacement for Elroy would be five characters in length and stored in a column of the same type.

Another statistic for columns is the percentage of NULL and non-NULL values. PostgreSQL tracks this as the `null_frac` or null fraction.

NULL values in columns consume less space but are still maintained in indexes.

When replacing text content, consider the column cardinality (*What Does Cardinality Mean in a Database?*³), frequency distributions, and the proportion of NULL values as you design your text replacement functions.

As you begin to identify sensitive columns like email that you want to scrub, how will you keep track of all the columns over time?

Tracking Columns With Sensitive Information

To maintain a list of columns with sensitive values, you can track that information within PostgreSQL.

PostgreSQL gives users the ability to add database-level comments to tables and fields. These comments are then dumped just like other structural details when `pg_dump` runs. You'll use the column commenting capability to track your sensitive columns.

3. <https://vertabelo.com/blog/cardinality-in-database/>

Connect to the `rideshare_development` database from `psql` and run this SQL statement:

```
sql/create_comment_for_table_column.sql
COMMENT ON COLUMN users.email IS 'sensitive_data=true';
```

The statement adds a comment to the `users.email` field using a simple key and value structure to mark a sensitive field.

To confirm the comment was added, run `\d+ users`. This expanded form of `\d` includes additional information like comments. If the output from `\d+ users` is difficult to read, try running `\pset format wrapped to toggle wrapping` and run again.

To the very right of the email column you should now see `sensitive_data=true` as the column comment.

For each sensitive field you've identified, create this type of comment.

To keep all databases in sync, create the database comments using a Rails Migration. Database comments will then be included in `db/structure.sql`.

You've now started to identify and track columns to scrub, but how will you perform the scrubbing?

Comparing Direct Updates and Table Copying Strategies

For your scrubbing system you'll use two techniques, "Direct Updates" and "Table Copying."

Direct Updates are simply SQL `UPDATE` statements that overwrites values. Direct Updates are recommended for smaller tables with fewer than one million rows. This is because `UPDATES` can become slow on very large tables.

One trade-off with Direct Updates is that database maintenance is required afterwards. Due to the design of PostgreSQL, an `UPDATE` statement for a row leaves around a former version that's now considered "dead." Dead row versions that haven't yet been cleaned up yet are part of what's referred to as *bloat* in a table or Index. Database maintenance that runs removes dead row versions and reduces bloat.

Database Maintenance operations may not be something you've run before. For Direct Updates, you'll run `VACUUM` and `REINDEX` commands as your maintenance operations.

A goal of a the Performance database scrubbing system is that it runs fast. With a fast running process, your team will have an easier time automating the steps and repeating the whole process periodically.

Because Direct Updates as a strategy is slow for large tables, you'll use a second for them called Table Copying. Table Copying could be for tables with more than 1 million rows. Table Copying as a strategy is based on a trick, where you replace the original table with a copy of the table. You'll see the steps in more detail in a later section.

A benefit of Table Copying is that it doesn't require the database maintenance that Direct Updates does.

Table Copying also runs faster by deferring some things like Index maintenance and constraint checks. To achieve that, Table Copying temporarily removes the constraints and Indexes, then adds them back at the end.

You've now seen a brief comparison of *Direct Updates* and *Table Copying* but haven't yet put these into practice.

How will you scrub the sensitive values?

Starting An Email Scrubber Function

To scrub sensitive values, you'll build up a database function that you'll use to replace sensitive values with non-sensitive ones.

For the first iteration of your scrubber function, you'll focus on the users table and email column from Rideshare. This column has a varchar type and the email addresses have a variety of domains like gmail.com. Your design goal is to preserve the email domain portion because it does not uniquely identify a person, but replace the first portion before the "@" symbol.

Create a database function called `SCRUB_EMAIL()`. As a Rails developer you may not have written database functions before. Don't worry, you'll be gradually introduced to it.

PostgreSQL functions can be written using SQL or a built-in Procedural Language called *PL/pgSQL*. The procedural language is available as an extension. Run `\dx` from psql to confirm that the `plpgsql` extension is listed.

SQL Functions are more limited but also less complex. PostgreSQL calls these Query Language (SQL) functions.⁴ Query Language Functions have some similar properties to Ruby methods you're familiar with like variable numbers

4. <https://www.postgresql.org/docs/current/xfunc-sql.html>

of arguments and default arguments. Functions written with PL/pgSQL can use variables, conditional branching, and loops.

Start with a skeleton structure without an implementation so that you can learn the basics.

The *signature* of the function accepts a single argument. The argument must be a varchar type and return a varchar text string.

Run psql from your terminal and type out this function structure:

```
sql/scrub_email_v1.sql
CREATE OR REPLACE FUNCTION scrub_email(email_address varchar(255))
RETURNS varchar(255) AS $$
SELECT
    email_address;
$$ LANGUAGE SQL;
```

CREATE OR REPLACE FUNCTION creates a named function or overwrites one that was defined with the same name and arguments. The function accepts an argument email_address with the type varchar(255). The argument type is important because your argument will be database column values with specific types, and the column type must match the accepted argument type exactly.

This function doesn't do anything yet. Notice it just calls SELECT on your input value.

Once you've pasted your function body into psql, you can now call your function using SELECT:

```
sql/select_scrub_email.sql
SELECT SCRUB_EMAIL(email) FROM users;
```

Nice! That's the basics of a SQL function.

Functions you defined are source code just like your Rails application code, so you'll want to keep them in version control and manage their versions. You could keep your functions in the Rails app in a directory called db/functions. From there you could manage changes, but the functions wouldn't automatically be added to your database.

Functions also have no built-in versioning concept. Wouldn't it be nice to have versioned functions and to have the functions automatically added to your database on deployment?

Fortunately the fx Ruby gem (*Versioned database functions and triggers for Rails*⁵) provides this functionality, and you'll put it to use. fx provides a Rails generator that generates a SQL file. This is where you place your function definition. Second, a Migration is generated to add the function.

The fx gem is already added to Rideshare so you can explore it there. An example of the fx function generator is below:

```
sh/rails_fx_gem_generate_function.sh
bin/rails generate fx:function scrub_email
```

The Migration in Rideshare looks like this:

```
ruby/migration_db_function.rb
class CreateFunctionScrubEmail < ActiveRecord::Migration[7.0]
  def change
    create_function :scrub_email
  end
end
```

fx provides a create_function that's similar to other Active Record Migration methods.

Functions are included in the application db/structure.sql file after running bin/rails db:migrate.

The first generation created a SQL file db/functions/scrub_email_v01.sql. Check out the db/functions directory in Rideshare to see revisions to the function.

Now that you've seen the basics of database functions and managing them with fx, you're ready to continue filling out your function implementation.

Implementing the Scrub Email Function

Your scrubber function implementation will build on top of built-in PostgreSQL functions.

Use the PostgreSQL SPLIT_PART()⁶ function to split a string into parts.

Pass an email address like SPLIT_PART(email_address, '@', 1) to return the first part before the @ symbol. The first part has a position of 1 (Not 0) so 1 is provided as the third argument.

Run the SPLIT_PART() example below in psql to get familiar with the function.

```
sql/select_split_part_function.sql
SELECT split_part('bob@example.com', '@', 1);
```

5. <https://github.com/teoljungberg/fx>

6. https://www.postgresqltutorial.com/postgresql-string-functions/postgresql-split_part/

```
-- split_part
-- -----
-- bob
```

Calling `SPLIT_PART()` returns `bob` which is visible in the commented out portion. `bob` is the text you'll replace with your function.

How long is `bob`? Use the built-in `LENGTH()` function. `bob` has a length of 3.

```
sql/select_length_function.sql
SELECT length('bob');
-- length
-- -----
--      3
```

Next you'll need some replacement text.

Use the built-in PostgreSQL MD5 hash string generation function to produce a string of characters as a data source.

The `MD5()` function generates 32-character strings from an input string. While you could use the original text as an input, MD5 hashed text is not considered secure cryptographically. The original text could be guessed by comparing hashed text to it.

For the Performance database purposes, it's OK to lose the original values. Use the `RANDOM()` function as input to `MD5()` and cast the output to `::TEXT`. `RANDOM()` provides a semi random input to `MD5()` that purposefully isn't connected to the original text.

The `MD5()` function generates long strings so you'll take a slice out that matches the length of your source email address. Typically this is going to be less than 32 characters for email portion before the "@" symbol.

One gotcha when generating MD5 text strings and comparing the first few characters is that there can be overlaps on those first few characters. You'll want to store unique values for the email address, so use a workaround.

The workaround is to make the generated replacement at least 5 characters long. Use the built-in `GREATEST()` function to use either the length of the original email address input text or a length of 5, whichever is greater.

Use the `CONCAT()` function to assemble the string parts, crafting a full replacement email address from three parts.

In the function implementation below, everything is brought together to replace an email address. You can overwrite the earlier version that didn't have a body by copying and pasting this function into `psql`. If you wish to manage

the function with Rideshare, you may need to drop the function from psql and then bin/rails db:reset to restore the original versions.

```
sql/scrub_email_function_full.sql
```

```
-- replace email_address with random text that is the same
-- length as the unique portion of an email address
-- before the "@" symbol.
-- Make the minimum length 5 characters to avoid
-- MD5 text generation collisions
```

```
CREATE OR REPLACE FUNCTION scrub_email(
  email_address VARCHAR(255)
) RETURNS VARCHAR(255) AS $$
SELECT
  CONCAT(
    SUBSTR(
      MD5(RANDOM()::TEXT),
      0,
      GREATEST(
        LENGTH(
          SPLIT_PART(email_address, '@', 1)
        ) + 1, 6
      )
    ),
    '@',
    SPLIT_PART(email_address, '@', 2)
  );
$$ LANGUAGE SQL;
```

Now that the function is defined, call it from a SELECT SQL statement as you did earlier from psql.

```
sql/scrub_email_short.sql
```

```
SELECT SCRUB_EMAIL('bob@gmail.com');
--      scrub_email
--  -----
--  5e9dd@gmail.com
```

Your function is the mechanism you'll use to replace the sensitive parts of email addresses with replacement text.

Try another email address that's longer and make sure the replacement text has the same length.

Run this SQL statement from psql.

```
sql/scrub_email_long.sql
```

```
SELECT SCRUB_EMAIL('bob-and-jane@gmail.com');
--      scrub_email
--  -----
--  39246b1ccf0a@gmail.com
```

With your scrubber function in place, the next phase in the development process is to run it for all rows.

How might you go about that?

Understanding Table Copying Trade-Offs

From this section forward, you'll focus on scrubbing table rows using the scrubber functions you've developed using Direct Updates and Table Copying.

In this section you'll dive into Table Copying. For Table Copying the first step is to create a new destination (or "target") table that is a clone of the original (or "source") table. The destination table will be empty with no data rows.

Run `psql --dbname rideshare_development` to connect to the `rideshare_development` database. Run the following command to copy the `users` table structure to a new table `users_copy`.

To do that you'll run the following SQL DDL statement from `psql`. This uses the `CREATE TABLE ... LIKE` table creation form. Run this statement:

```
sql/table_copying_create_like.sql
```

```
CREATE TABLE users_copy (LIKE users INCLUDING ALL);
```

You'll now have a `users_copy` table that's a clone of the original `users` table with no data rows. Explore it with `\d users_copy`. Confirm that there are no rows by running `SELECT COUNT(*) FROM users_copy`.

The reason you didn't fill the table rows yet is very important and you'll find out why very soon.

If you have a messy database, remember that you can reset back to the original state first and download the data dump. Alternatively you can start from an empty database and run the following commands to generate new sample data.

```
bin/rails db:reset
```

```
bin/rails data_generators:generate_all
```

You should now have data rows that are ready to be scrubbed.

Review the general form of the `INSERT` statement below, which takes a the query results of a `SELECT` statement as input. By connecting these two statements you'll copy rows from one table to the other. Since the target table has all the database objects including indexes, constraints, defaults, triggers, and more, the insert process is slower than it would be without all of those objects.

You don't need to run this statement now, you're just learning the format.

```
sql/table_copying_insert_into.sql
```

```
INSERT INTO users_copy(
    first_name,
    last_name,
    email,
    type,
    created_at,
    updated_at
)
(
    SELECT
        first_name,
        last_name,
        email,
        type,
        created_at,
        updated_at
    FROM users
);
```

You're now ready to bring a couple of concepts together.

As you did earlier with single SELECT statements, use a SELECT statement and your scrubber function to transform values on the fly before they're inserted.

Run `\df scrub*` in `psql` to make sure your scrubber function is listed.

The `SCRUB_EMAIL()` function is used with the value of the `users.email` column. Run the following command below to insert scrubbed rows.

```
sql/scrubbing_on_the_fly.sql
```

```
INSERT INTO users_copy(
    id, first_name, last_name,
    email, type, created_at, updated_at
)
(
    SELECT
        id, first_name, last_name,
        SCRUB_EMAIL(email), -- scrubber function
        type, created_at, updated_at
    FROM users
);
```

The `users_copy` table should now be filled with scrubbed rows! Compare the same row from the `users` table and `users_copy` table.

```
sql/scrubbed_row_comparison.sql
```

```
SELECT email FROM users WHERE id = 1;
      email
-----
Beau-Marvin-driver-0@email.com
```

```
SELECT email FROM users_copy WHERE id = 1;
      email
-----
16acaa1aa55e432d69ea@email.com
```

Although this technique was fast, you can speed things up further. Speeding this process up is important when you're working with millions of rows of data. How might you go about that?

Speeding Up Inserts For Table Copying

By using database functions inside PostgreSQL to perform scrubbing, you're removing latency that would otherwise be added to the process when the client application communicates with PostgreSQL.

Although running the process is fast, it can be made even faster with more table manipulation work. To do that, you'll use some tricks that are well known to speed up row copying.

A note upfront. These tricks speed up the Insert rate but add significant complexity to the process. Try keeping things simple if you're building your own scrubbing system to start, and then use these tricks only if you need to make the process faster.

The tricks rely on removing database constraints and indexes on the destination side of copying. They aren't really removed, but they're intentionally skipped initially and then added after the row copying has completed. To achieve this you'll be doing a lot of direct manipulation of table objects which is a lot to learn! You'll gradually build up your comfort with these techniques if they're new.

To get started, drop the `users_copy` table you created earlier since you're going to create it again. You'll be working in `psql` for this section. Earlier you created the `users_copy` table using the `INCLUDING ALL` keywords but this time you'll do something different.

Create the table with the keywords `INCLUDING ALL EXCLUDING INDEXES`:

```
sql/table_create_like_including_all_excluding_indexes.sql
DROP TABLE IF EXISTS users_copy;

CREATE TABLE users_copy (LIKE users INCLUDING ALL EXCLUDING INDEXES);
```

Explore the new version of `users_copy` with `\d users_copy`. This version has many of the database objects before, but the table Indexes are missing as well as the Primary Key constraint on the `id` column.

Run the `sql/scrubbing_on_the_fly.sql` Insert statement from earlier. If you've enabled timing and compared the time to running it with Indexes, it should be faster without them.

Besides Indexes, you can defer creating Foreign Key and Primary Key constraints, and manipulate the Sequence ownership and assignment, in addition to deferring Index creation.

Let's work on handling Foreign Key Constraints. You don't need to know how they work much now, they'll be covered more in a future chapter. For now you're just manipulating them as objects in the database.

Since the `users` table does not have Foreign Key constraints, use a different table in Rideshare to explore this.

From `psql`, run the following SQL statement to list all Foreign Key Constraints in the `public` schema, taking note of the ones that are listed for the `trips` table. `trips` is another Rideshare table. This query uses the `pg_constraint`⁷ System Catalog:

```
sql/list_all_constraints.sql
-- list constraints in 'public' schema
SELECT
    conrelid::regclass AS table_name,
    conname AS foreign_key,
    PG_GET_CONSTRAINTDEF(oid)
FROM pg_constraint
WHERE contype = 'f'
AND connamespace = 'public'::regnamespace
ORDER BY conrelid::regclass::text, contype DESC;
```

From the `trips` table, select the Foreign Key Constraint named `fk_rails_e7560abc33`. The constraint name was generated by Active Record and the name may change from what's here. Refer to the latest copy of Rideshare source code. The constraint covers the `driver_id` field on the `trips` table and refers to the `users.id` column.

To restore the constraint after copying has completed, you'll add it to the `users_copy` table. You'll want the exact same definition and name.

Creating Constraints manually can get very tedious though, and there could be dozens of constraints to create!

An improved technique for listing them is to list them as a creation DDL statement, making it easier to copy and paste the statements.

7. <https://www.postgresql.org/docs/current/catalog-pg-constraint.html>

To do that, run psql and enter the following query. Supply the constraint name from above. The result will be the constraint definition, showing which table and columns it covers.

```
sql/constraint_definition_ddl.sql
```

```
SELECT
  FORMAT(
    'ALTER TABLE %I.%I ADD CONSTRAINT %I %S;',
    connamespace::regnamespace,
    conrelid::regclass,
    conname,
    PG_GET_CONSTRAINTDEF(oid)
  )
FROM
  pg_constraint
WHERE
  conname IN ('fk_rails_e7560abc33');
```

You'll need a trips_copy table to test this on, similar to the users_copy table above. Run the following statements to create a trips_copy table. With that in place, run the ALTER TABLE statement below from the query above to add the Foreign Key Constraint.

```
CREATE TABLE trips_copy (
  LIKE trips INCLUDING ALL EXCLUDING INDEXES
);

ALTER TABLE public.trips_copy
ADD CONSTRAINT fk_rails_e7560abc33 FOREIGN KEY (driver_id)
REFERENCES users(id);
```

You've now seen one way to add the constraint to the destination table. You can DROP TABLE trips_copy; to clean things up.

Next you'll consider how to handle Sequences used for the Primary Key column. Note that this table is not using an Identity column (See: *Serial type versus Identity columns*⁸) and is using a Sequence for Primary Key column values.

The users table id column is the Primary Key and has a unique value enforced by the Primary Key Constraint, handed out from a Sequence. The Sequence is another database object and has the name users_id_seq which follows the conventional naming format for Sequences from Active Record tables.

Just like with Foreign Key Constraints you may list out all the Sequences in the database. Run the following "List all Sequences" SQL query⁹ from psql.

8. <https://wanago.io/2022/02/21/serial-type-identity-columns-postgresql-typeorm/>

9. https://github.com/andyatkinson/pg_scripts/blob/master/list_all_sequences.sql

```
sql/list_sequences_table_column_owner.sql
```

```
SELECT
    s.relname AS seq,
    n.nspname AS sch,
    t.relname AS tab,
    a.attname AS col
FROM
    pg_class s
JOIN pg_depend d ON d.objid = s.oid
AND d.classid = 'pg_class'::regclass
AND d.refclassid = 'pg_class'::regclass
JOIN pg_class t ON t.oid = d.refobjid
JOIN pg_namespace n ON n.oid = t.relnamespace
JOIN pg_attribute a ON a.attrelid = t.oid
AND a.attnum = d.refobjsubid
WHERE
    s.relkind = 'S'
    AND d.deptype = 'a';
```

Since there is no activity on the users table right now and since you're working on a replacement table called users_copy, you'll change the ownership of the sequence so that it's owned by the users_copy table. This is because eventually the users_copy table will take over for users.

Warning: you wouldn't want to modify the sequence this way in a live system that was being used by a users table creating new rows. The Sequence must be in the same Schema as the table and the table owner must be the same. users_id_seq is in the Public schema. Check the owner of users and users_copy with the \dt meta command. If the owners are different, to change the owner of users_copy to postgres, for example, run this statement:

```
ALTER TABLE users_copy OWNER TO postgres;
```

Once the owner is the same for both users and users_copy, run the following statement to set the ownership for the sequence.

```
ALTER SEQUENCE users_id_seq OWNED BY users_copy.id;
```

The sequence is now in place for the users_copy table.

To see the Indexes for a table from psql, use the \d users meta command. The indexes are at the bottom, but they aren't listed in a way that makes them easy to create again.

Use the same trick you used earlier to list the indexes as DDL create statements.

Run the following SQL query to do that:

```
sql/list_users_table_indexes.sql
```

```
SELECT pg_get_indexdef(indexrelid) || ';' AS index
FROM pg_index
WHERE indrelid = 'public.users'::regclass;
```

Rideshare has four indexes on the users table. Take a look at the Unique index on the email column.

```
CREATE UNIQUE INDEX index_users_on_email
ON public.users USING btree (email);
```

The table name is “schema qualified” and shows the public schema name. Change the table name from users to users_copy. You’ll also need to make the name unique since index_users_on_email already exists for the users table. Add something unique on the end like a number.

Repeat these general steps for the other three indexes.

You’ve now have a technique to bring Foreign Key Constraints, Primary Key column Sequence, and Indexes forward to a copied table.

The users_copy table can now “take over” from the users table since it has a matching design. The last step in this process is to rename the users_copy table to users. This combination of Drop and Rename should be performed inside a transaction so it happens together.

From psql, run the following SQL statements to DROP the source table and rename the destination table.

```
sql/finalize_table_copying_users.sql
```

```
BEGIN;
-- drop the original table and related objects
DROP TABLE users CASCADE;

-- rename the destination table to be the source table name
ALTER TABLE users_copy RENAME TO users;

COMMIT;
```

Note that CASCADE was used to drop all related objects. Be careful using CASCADE in your system. Rideshare has more objects connected to the users table, like Views and Materialized Views that are being discarded with CASCADE. For now it’s OK to remove them. We know that we can always reset the Rideshare schema by running bin/rails db:reset if we want to start over.

From the perspective of the Rails application, the new table works as-is with the Driver and Rider model classes since it has the same design. In a real application you wouldn’t even need to restart your Rails application because

the schema has not changed, although restarting is usually a good idea to clear out any cache related issues.

You’ve now seen the Table Copying process and run through it end to end.

Moving all of the database objects around is a lot of work.

For smaller tables, directly updating rows will be a more straightforward technique. You’ll work on that next.

Using Direct Updates For Text Replacement

Direct Updates modifies the users table row data in place. This strategy is considered “destructive” compared with Table Copying strategy, because it doesn’t provide a “rollback” mechanism. That’s ok for your purposes here but it’s worth noting that difference.

Run the following UPDATE statement from psql to scrub email address columns for every row.

```
sql/direct_updates_users.sql
```

```
UPDATE users
SET email = SCRUB_EMAIL(email);
```

Running the statement above should show something like below, indicating all rows were updated.

```
UPDATE 20300
```

If you were running this on hundreds of thousands of rows, you may wish to update a batch of, say, 10,000 rows at a time. Working on batches of rows will be covered in an upcoming section. For now, you’ll use single statements.

Notice how with Direct Updates, you don’t need to worry about managing Foreign Key Constraints, Sequence modifications, or Index creations as you did with Table Copying.

On the other hand, performing large amounts of Updates can contribute to degraded performance. You’ll learn how to perform some maintenance after bulk updates to keep performance optimal.

What does that look like?

Performing Database Maintenance

Because you’ve created a large amount of Updates, new row versions replaced old ones, and those old ones will no longer be referenced. Remember that row versions are immutable in PostgreSQL. The former version is a “dead” row

when it's no longer referenced by any live transaction. The space it consumes contributes to "Bloat" and could be reclaimed for other purposes. *Autovacuum* manages this for you keeping things tidy and runs automatically. Autovacuum runs in the background as a scheduler system and runs VACUUM operations for each of your tables.

While Autovacuum is great, for manual operations like Direct Updates, it's a good idea to immediately perform an equivalent operation manually following the batch of Updates. This means you'll run a manual VACUUM operation for each table you've updated data in.

To run a manual VACUUM, from psql run this statement:

```
VACUUM (ANALYZE, VERBOSE) users;
```

The ANALYZE and VERBOSE keywords here are arguments to the VACUUM command.

ANALYZE updates table statistics that are collected for the table.

VERBOSE adds more information showing what's happening, listing the number of pages being scanned, live rows, and dead rows.

Besides Vacuum, you'll also want to refresh your Indexes following a big Update. This operation is called *Reindexing*. Although the command implies the Index is updated in place, Reindexing actually replaces the original Index with a freshly built on. To rebuild indexes you'll use the REINDEX command.

For PostgreSQL version 12, the REINDEX command gained a new option called CONCURRENTLY. This was very significant because it makes it possible to rebuild indexes without stopping PostgreSQL first.

For the Performance database, it's being refreshed offline. The REINDEX command can be run without CONCURRENTLY because there are no other concurrent users. This makes the operation run faster compared with CONCURRENTLY.

Practice rebuilding the index `index_users_on_email` by running this statement from psql:

```
REINDEX INDEX index_users_on_email;
```

You've now performed VACUUM, ANALYZE, and REINDEX operations following a big update. These maintenance operations will help restore optimal performance for the table.

Earlier you learned that large amounts of Updates might be performed in batches, instead of being performed as one single big UPDATE statement.

Imagine that you'd like to break up the operation into batches using PostgreSQL capabilities. How might you do that?

Performing Modifications In Batches

Earlier you learned how to write *Functions* in SQL, and that they can also be written using the PL/pgSQL Procedural Language.

PostgreSQL offers another capability that extends functions further called *Procedures*, introduced in version 11. You'll use a Procedure to call the Update function you developed earlier, and add some batching around it.

Procedures can use loops like you're used to when working in Ruby.

To create a range of rows to work with, you'll use the Primary Key id column. A variable sets the batch size, which is 1000 by default.

Read through the implementation below. The parts will be explained.

```
sql/scrub_batched_direct_updates.sql
CREATE OR REPLACE PROCEDURE scrub_batches()
LANGUAGE plpgsql
AS $$
DECLARE
    current_id INT := (SELECT MIN(id) FROM users);
    max_id INT := (SELECT MAX(id) FROM users);
    batch_size INT := 1000;
    rows_updated INT;
BEGIN
    WHILE current_id <= max_id LOOP
        -- the UPDATE by `id` range
        UPDATE users
        SET email = SCRUB_EMAIL(email)
        WHERE id >= current_id
        AND id < current_id + batch_size;

        GET DIAGNOSTICS rows_updated = ROW_COUNT;

        COMMIT;
        RAISE NOTICE 'current_id: % - Number of rows updated: %',
            current_id, rows_updated;

        current_id := current_id + batch_size + 1;
    END LOOP;
END;
$$;

-- Call the Procedure
CALL scrub_batches();
```

First, a WHILE loop is used to run until a condition is met.

The `GET DIAGNOSTICS rows_updated = ROW_COUNT;` clause captures the number of rows being updated in one loop iteration. The `UPDATE` statement updates all of the rows that are matched inside of the `id` column range. This will update up to 1000 rows if there are no gaps.

Values are printed out in a similar way to puts in Ruby by using the `RAISE NOTICE` clause. The `%` symbol is used in a similar way to String interpolation in Ruby.

Try copying and pasting the `CREATE PROCEDURE` block into `psql`. This defines the procedure but doesn't call it. The last line `CALL SCRUB_BATCHES();` is what calls the procedure.

If you ran the procedure you'll see similar output to what's below. The `current_id` is printed starting from the `min(id)` value and it starts updating rows until the `MAX(id)` is reached.

```
NOTICE: current_id: 1 - Number of rows updated: 1000
NOTICE: current_id: 1002 - Number of rows updated: 1000
NOTICE: current_id: 2003 - Number of rows updated: 1000
```

You've now seen a technique to perform `UPDATE` statements in batches, using a capability that's built in to PostgreSQL.

What's next?

What's Next For Your Performance Database

In this chapter you've saw how you could build a scrubber system for your production data. By creating a sanitized copy of your production database, you can unlock meaningful Performance testing for your team, and create a safe place to try out destructive modifications since it can be periodically reset.

You compared Direct Updates and Table Copying strategies and learned about their trade-offs.

By running the Performance database on an instance with similar power to your production instance, you'll have a great simulation environment for your team.

In the next chapter you'll expand on the Foreign Key constraints and other constraint types you started working with. You'll dive into how you can improve the correctness and consistency of the data in your database, while managing trade-offs like how performance is impacted.

Read on to learn more.

Data Correctness and Consistency

In this chapter you'll use Constraints to improve your data layer. PostgreSQL offers a variety of constraint types that help you make sure your data is correct and consistent.

Constraints are objects you add to your columns, tables, or even for multiple tables. In most Ruby on Rails applications, you'll also validate data at the application layer using Active Record Validations.

Do you need both? Database constraints and Application level data validation are complementary so you'll use both.

To help understand what consistent data looks like and how constraints enforce consistency, you'll start with an example. One purpose constraints serve is to enforce *Referential Integrity* between rows in two tables.

When Referential Integrity is violated it means that one half of the relationship is missing. Missing data results in an application bug and data related bugs can be a common type of bug due to invalid data input.

To make sure your data is referentially intact, you'll add Foreign Key constraints. Foreign Key constraints work together with Primary Key constraints on a table that's referred to, to establish and enforce a data relationship.

There must not be violations of the constraint when it's added, or it will fail to be added. Does this mean you can't add constraints to an existing database with invalid data? You'll see how to handle this.

Besides invalid data, for large and busy databases, there are operational challenges to safely adding them to production databases. For this reason, developers may avoid adding them. However, you'll see how to handle these scenarios and add constraints even for big and busy databases.

Correctness Terminology



- Constraints — Database object used to constrain data and enforce correctness
 - Correctness — Ensuring that data is consistent with the constraint definition
 - Referential Integrity — Ensuring rows in two tables with common values both exist
 - Enum — Database list object to constrain possible field values
 - Domain — Like an enum but can be shared across tables
-

PostgreSQL provides a variety of constraint types¹ that will be covered in this chapter.

- PRIMARY KEY
- NOT NULL
- UNIQUE (covered in [on page 56](#))
- REFERENCES (Foreign Key)
- CHECK
- EXCLUDE (Exclusion)

First up is the UNIQUE constraint.

Multiple Column Uniqueness

One of the interesting challenges in data modeling is determining whether or which conditions for a table make rows unique. Constraints are used to describe this property, and enforce it.

The definition of a unique row is usually driven from a single column, or multiple columns. The definition can also change over time. For example an application that expands geographically, might have City names as unique within a single state. What about when it expands to more U.S. States? “Portland” could refer to Portland, Maine or Portland, Oregon. At that point, you’d introduce a State concept and change the definition of uniqueness to be the combination of city name and state name.

In Rideshare there is a locations table with city and state columns. The table has a UNIQUE constraint on the address column.

1. <https://www.postgresql.org/docs/current/ddl-constraints.html>

You'll add a new unique constraint that defines uniqueness on both City and State.

Unique constraints can be added directly to a table. PostgreSQL automatically adds an index for a new unique constraint, to make enforcement fast.

For a running system, adding the constraint via a Unique index is more common. A Unique constraint is added to the table. This means the table will need to be locked to check that all rows are compatible with the constraint definition.

Because this would cause a disruptive lock, it is more common to add the constraint via an Index. With the Index technique, the CONCURRENTLY keyword can be used which minimizes the lock period on the table. Concurrently is a good default for any live system, but does make the operation take longer. With Concurrently, concurrent transactions can continue to run while the Index is built.

Let's dive into this example.

Review the following Migration from Rideshare.

The `unique: true` option set in Ruby makes it a UNIQUE index and the `algorithm: :concurrently` options adds the CONCURRENTLY keyword to the Index creation DDL.

```
class AddLocationsCityStateUnique < ActiveRecord::Migration[7.1]
  disable_ddl_transaction!

  def change
    add_index :locations, [:city, :state],
      unique: true,
      algorithm: :concurrently
  end
end
```

What about when there are duplicate city names? In this case PostgreSQL won't allow the constraint to be added. You'll need to address the violations before it can be added, but you might not be sure what to do.

Fixing Constraint Violations

When adding constraints like NOT NULL or FOREIGN KEY, all existing rows in the table are checked. When the table has millions of rows, this might take a while to add. Querying batches of data to identify possible violations in Ruby code could take a while.

How can you ensure there are no violations in advance?

To do that, you'll need to query for possible violations, but do it efficiently since your table may have millions of rows.

In the SQL For Devs post *Delete Duplicate Rows*,² Tobias describes two methods to identify duplicates and delete them in a single efficient query.

Two approaches are given, a simpler and more complex approach. In the simpler approach, rows are queried and grouped by the columns that make them unique, and then rows greater than the MIN(id) or less than the MAX(id) are deleted. Remember that the goal is to remove duplicate rows so that the constraint can be added. If your duplicates aren't rows you can remove, you may want to modify them instead of deleting them.

In the more complex form, a couple of advanced concepts are used, and the additional complexity offers more flexibility on which results to delete.

This technique uses a Common Table Expression (CTE) for the main query. Result rows get a value from the ROW_NUMBER() *Window Function*³ which uses the PARTITION BY option. This partitions (or "segments") the results based on the columns that define uniqueness. CTEs and Window Functions are not yet concepts you've worked with, but for now the goal is more on how to work with the query results.

Each result row in the results gets a row number. The rows can be ordered in different ways to get a different row number. The rows are deleted based on a row number condition, such as rows > 1. This is a good general pattern if you find that you need to efficiently remove duplicates before adding a new Unique constraint. Remember that once the duplicates are modified or removed, you'll then be able to add the Unique constraint.

You've now covered some basics of Unique constraints, including how to add them, and some of the challenges when the definition of Uniqueness changes.

Next you'll look at FOREIGN KEY constraints, learning how to use them to improve your data model.

Enforcing Relationships With Foreign Keys

You've described data relationships between Active Record models in model code. The enforcement of these relationships can be achieved to an extent using *Validations* with hooks into lifecycle events like creating, updating, or deleting a row. The paper *Feral Concurrency Control: An Empirical Investigation*

2. <https://sqlfordevs.com/delete-duplicate-rows>

3. <https://www.postgresql.org/docs/current/tutorial-window.html>

of *Modern Application Integrity*⁴ explores how Active Record models use Associations and Validations within the Ruby application code layer.

Ruby on Rails developers might think of client applications being JavaScript front-ends. Consider how Active Record applications are client applications from the perspective of a PostgreSQL server. Active Record Validations while useful and commonly used, will always have weaker guarantees in their ability to enforce data consistency compared with enforcement within the PostgreSQL database server. PostgreSQL is designed for high concurrency and offers transactions, and transaction isolation, to ensure client Inserts, Updates, and Deletes, are consistent with the constraints that are defined.

Let's explore describing and enforcing referential integrity in the database. The general technique is that a column (usually ends in `_id`) stores the value of the Primary Key (usually `id`). This is a one-way relationship. The table referred to is a dependency.

Foreign Key constraints are not added by default so it's up to developers to add them.

In the scenario above, you'll add a Foreign Key constraint that lists both columns and links them. This will give you a strong guarantee of referentially intact data. Once the Foreign Key constraint is in place, PostgreSQL will prevent a client from deleting either row.

For many years Rails did not support database Foreign Key constraints. Native support for database Foreign Keys was added to Rails Version 4.2 released in 2014.

Let's review a Foreign Key⁵ in Rideshare.

When a Driver accepts a Trip Request, a Trip is created. Trips depend on Trip Requests so a constraint will be added to describe and enforce this relationship.

Review the SQL below which added the Foreign Key constraint.

```
sql/foreign_key_constraint_trips.sql
ALTER TABLE ONLY trips
ADD CONSTRAINT fk_trips_trip_requests
FOREIGN KEY (trip_request_id)
REFERENCES trip_requests(id);
```

4. <http://www.bailis.org/papers/feral-sigmod2015.pdf>

5. https://guides.rubyonrails.org/active_record_migrations.html#foreign-keys

Duplicate Constraints and Indexes



Constraints and Indexes can be duplicated when they have unique names. Be careful not to add duplicate Constraints or Indexes because they consume space and add unnecessary latency to write operations. If you find a duplicate, one of them should be removed.

You've now seen how to use Foreign Key Constraints to describe and enforce data relationships between tables.

Next you'll work with the CHECK constraint, a versatile type that can be used for multiple purposes.

The Versatile Check Constraint

A CHECK constraint provides more flexibility than checking for NULL values, or linking tables together. When you create a Check constraint, the constraint describes and enforces a boolean condition that you've written using SQL. CHECK constraints are added to a table.

For Rails developers, you may now be using Check constraints since they've only recently gained more support in Active Record. Think of Check Constraints as similar to Active Record Model Validations, but offering stronger guarantees by being defined in the database.

Check constraints can be used with Active Record Validations. You'll see an example later in this section of a boolean condition in the constraint definition.

Check constraints have a valuable capability as well, that allows them to be put in place only for new row modifications, which are new Inserts, Updates, or Deletes.

UNIQUE and NOT NULL constraints do not have this capability. When those constraint types are added to a table, all rows are checked for compatibility. This makes them riskier to add to large and busy tables.

What can you do then for an existing database when you discover places where you'd like to add a Unique or Not Null constraint?

The answer is that you can use Check constraints as a transitional element. How is this accomplished? To do this, you'll add a CHECK constraint and mark it NOT VALID. Foreign Key constraints also support the NOT VALID option. With that in place, you'll be able to then validate all the earlier rows using a less disruptive operation. Once you've validated all rows, you'll be able to replace the Check constraint with the desired Unique or Not Null constraint.

This process does involve a few steps, so you'll want to practice it. To practice these steps, you can review or find some opportunities within Rideshare. The following examples were inspired by *Postgres Constraints for Newbies*⁶ from Crunchy Data and have been adapted to Rideshare.

You'll add Check constraints to the Vehicle Reservations table.

To get started, take a look at the Check constraints on the trips table.

When a Trip is completed, the completion time is set in the `completed_at` column. The completion time should always occur after the trip was created, stored in the `created_at` column. This “chronological consistency” can be described and enforced using a Check constraint.

Rails 6.1 natively supports Check constraints in Active Record syntax.⁷ Support for `if_exists`⁸ was added later to make it easier to add the Check constraint only when it doesn't already exist.

An Active Record Migration to add this constraint would look like this:

```
ruby/migration_trips_check_constraint.rb
```

```
class AddCheckConstraintTripsCompletedAt < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :trips, "completed_at > created_at",
      name: "trips_completed_after_created"
  end
end
```

This Migration will create SQL like below.

```
sql/check_constraint_trips.sql
```

```
ALTER TABLE trips
ADD CONSTRAINT trips_completed_after_created
CHECK (completed_at > created_at);
```

The Vehicle Reservation start and end times should also make sense chronologically. This is a Check constraint opportunity on the `vehicle_reservations` table that has not been added. See if you can add it, referring to the example above.

If there are rows that violate the constraint, PostgreSQL will display an error message:

```
sql/error_constraint_violation.sql
```

```
ERROR: check constraint "trips_completed_after_created"
```

6. <https://www.crunchydata.com/blog/postgres-constraints-for-newbies>

7. <https://blog.saeloun.com/2021/01/08/rails-6-check-constraints-database-migrations>

8. <https://github.com/rails/rails/commit/25f97a66bdae6efe788b2d0ab7ab9cef6fc5a23a>

of relation "trips" is violated by some row

You've now seen how Check constraints can help make sure your timestamps are in the correct order. Having constraints like this in place might help you avoid a data related bug in your application. It also provides a form of documentation within the database without needing client application code.

Chronologically consistent timestamps are relatively straightforward. What about a constraint that's a bit more complex?

Imagine that Vehicle Reservations must be at least 30 minutes long.

This requirement can be described and enforced as a Check constraint. Review the definition below. This builds on what you did earlier, making sure the `ends_at` column is at least 30 minutes after the `starts_at` value. Both of these columns also disallow Null values using NOT NULL constraints.

Run the following SQL statement in psql:

```
sql/check_constraint_reservation_length.sql
ALTER TABLE vehicle_reservations
ADD CONSTRAINT vehicle_reservation_minimum_length
CHECK (ends_at >= (starts_at + INTERVAL '30 minutes'));
```

Earlier you learned about a second purpose for Check constraints, to use them as a transitional element. To do that the constraint is initially defined as NOT VALID. When the constraint is defined this way, it is applied only to new row modifications.

Constraint enforcement can also be *deferred* which is slightly different, and has a specific meaning related to Transactions. Read on to the next section to learn more.

Deferring Constraint Checks

As you've seen, constraints enforce a definition you've created. Enforcement happens at the time rows are Inserted, Updated, or Deleted.

PostgreSQL gives you the option to defer when the enforcement happens. UNIQUE, PRIMARY KEY, Foreign Key (REFERENCES), and Exclusion (EXCLUDE) constraints support this *deferrable* capability. However, none of them are deferred by default, they are "checked immediately."

If you wish to make the constraint deferrable, you'll need to create it that way from the beginning, or replace it setting the DEFERRABLE option as part of the creation definition.

The deferrable behavior can be used as needed, or be the default behavior. To use it as needed, use the SET CONSTRAINT⁹ operation within an explicit transaction you've created.

You might be wondering about examples where deferring the constraint enforcement is helpful. In the Hashrocket post *Deferring Constraints in PostgreSQL*,¹⁰ the author shows how Unique list item positions can be shuffled around by deferring constraint enforcement.

Since list items have unique position values, to re-order list items, there would temporarily be duplicate position values.

To achieve this, updating the list item positions are all done within a single transaction. The Unique position value checking and enforcement is deferred until the transaction commits. Since all positions are Unique by the end, the transition will commit successfully.

If this behavior makes sense as the default, you can define your constraint using DEFERRABLE INITIALLY DEFERRED. This makes the constraint enforcement within a transaction deferred by default.

You've now seen how to control when constraints are enforced, and how that's useful.

Next you'll explore another constraint type. The Exclusion (EXCLUDE) Constraint is powerful and less commonly used. This constraint type can be used with multiple rows within the same table. How does that work?

Preventing Overlaps With An Exclusion Constraint

An Exclusion constraint works on multiple rows of the same table. This is used to describe how rows should or should not relate to each other.

What's an example? In Rideshare, Vehicle Reservations should not overlap. Since reservations are stored as rows, you can use an Exclusion constraint to describe and enforce this definition.

Two Riders should not be able to reserve the same vehicle at the same time. Typically this business logic will be written in the application. Since this requirement is very important, you'd like to leverage the stronger consistency guarantee that an Exclusion constraint can offer.

9. <https://www.postgresql.org/docs/current/sql-set-constraints.html>

10. <https://hashrocket.com/blog/posts/deferring-database-constraints>

Overlapping Vehicle Reservations would cause a bad user experience. Not only would the reservation be impossible to fulfill for both people simultaneously, it would be unclear which reservation to honor. The first one? The most recent one? One of your customers will be upset.

That's something you'd like to avoid.

To create the Exclusion constraint you'll use a GiST index provided by the `btree_gist` extension, which you'll enable in your database. You haven't yet worked with GiST indexes; for now it's enough to know that they are the Index type used for your Exclusion constraint.

Review and run the SQL below to add this constraint. If the constraint exists it will first be removed. Each of the pieces will be explained in the next section.

```
sql/exclusion_constraint.sql
```

```
ALTER TABLE vehicle_reservations
DROP CONSTRAINT IF EXISTS non_overlapping_vehicle_reservation;

CREATE EXTENSION IF NOT EXISTS btree_gist;

ALTER TABLE vehicle_reservations
ADD CONSTRAINT non_overlapping_vehicle_reservation
EXCLUDE USING gist (
    vehicle_id WITH =,
    TSTZRANGE(starts_at, ends_at) WITH &&
)
WHERE (NOT canceled);
```

Let's break down what's happening.

The constraint is named `non_overlapping_vehicle_reservation`. The `btree_gist` extension is enabled because a GiST index is needed.

Identical `vehicle_id` values are selected using the `WITH =` clause and operator.

For the same `vehicle_id`, the `starts_at` and `ends_at` column values are compared using the `WITH &&` clause and operator.

The `&&` operator is a special one that identifies overlaps comparing two timestamp values using the `TSTZRANGE` Range type. Overlaps here mean reservations made with overlapping time ranges.

The last part is a `WHERE` clause with `WHERE (NOT canceled)`. This clause excludes canceled Vehicle Reservations from the definition. You wouldn't want canceled reservations to prevent new reservations from being added, so that your business could still collect revenue for that time period.

And there you have it.

The Exclusion constraint is a more complex but once you're familiar with the syntax, expressive and powerful constraint type you can use for conditions across multiple rows of the same table.

This concludes exploring database constraint types. In the next section you'll switch gears into exploring Active Record application Validations.

Active Record provides flexible and useful built-in Validation types. Sometimes these aren't enough for your custom requirements. Fortunately Active Record allows you to write Custom Validators that can be combined with the built-in Validations. How can you do that?

Creating Active Record Custom Validators

Custom Validators¹¹ are created using some object oriented design concepts.

A Ruby class that extends the `ActiveModel::Validator` class and implements a `validate()` instance method is a Custom Validator class. The `.validate()` method accepts a record argument.

Individual attributes for a model can be validated by extending the `ActiveModel::EachValidator` class, and implementing the `.validate_each()` method. This method has a signature of `validate_each(record, attribute, value)`, allowing you to access the record to validate, then the attribute by name and it's value.

Review the Rails Guides section on Custom Validators called *Performing Custom Validations*. The email validator from Rails Guides has been copied below as an example.

```
ruby/email_validator.rb
class EmailValidator < ActiveModel::EachValidator

  EMAIL_REGEXP_FORMAT = /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i

  def validate_each(record, attribute, value)
    unless value =~ EMAIL_REGEXP_FORMAT
      record.errors.add(
        attribute,
        (options[:message] || "is not an email")
      )
    end
  end
end
```

11. https://guides.rubyonrails.org/active_record_validations.html#performing-custom-validations

To use the validator, the `email: true` configuration is added to a model attribute that you want to validate. Model attributes support multiple validations, allowing you to add both built-in ones and Custom Validators.

First you'll define the `EmailValidator` class, and then you'll use it. What does that look like? The `EmailValidator` class was added to Rideshare and is used in the `User` model for the email field.

After studying this Custom Validator example, it's time for you to make your own.

Imagine you're developing a feature for Rideshare that must validate Driver's License numbers. You've got a Driver's License from the State of Minnesota in the United States to validate. How would you do that?

Driver's Licenses in the United States have different formats in each state. Some states share their formats.

According to *U.S. Driver's License Numbers*,¹² Minnesota Driver's License numbers have one letter and 12 numbers. For example, `P800000224322` starts with the letter "P" and then has 12 numbers.

A Regular Expression of `[a-zA-Z]\d{12}` is even provided on the website for reference. You can use this Regular Expression as the core part of your custom validator implementation.

A skeleton is provided below. Before reading on, try and fill it out using the example above. Once you're done, continue reading.

```
class DriversLicenseValidator < ActiveRecord::EachValidator
  def validate_each(record, attribute, value)
    end
end
```

A completed example for Driver's License Numbers for Minnesota has been added to the `app/validators` directory, and is copied below:

`ruby/drivers_license_number_validator.rb`

```
class DriversLicenseValidator < ActiveRecord::EachValidator

  DL_MN_REGEXP_FORMAT = /[a-zA-Z]\d{12}/i
  DEFAULT_MESSAGE = "is not a valid driver's license number"

  def validate_each(record, attribute, value)
    unless value =~ DL_MN_REGEXP_FORMAT
      record.errors.add(
        attribute,
        options[:message] || DEFAULT_MESSAGE
      )
    end
  end
end
```

12. <https://success.myshn.net/Data-Protection/Data-Identifiers/U.S.-Driver%27s-License-Numbers>

```

    )
  end
end
end

```

Your Custom Validator can be combined with built-in Validations like presence and uniqueness.

The Driver model in Rideshare combines these validators:

```

ruby/driver_model.rb
class Driver < User
  validates :drivers_license_number,
    presence: true,
    uniqueness: true,
    drivers_license: true
end

```

You’ve now seen how multiple Active Record validations can be combined. Pretty slick. Your validations help enforce that data is populated, unique, and matches an expected format.

Although these rules are rigid; at other times you may want more flexibility.

One example is case sensitivity. Email addresses are not case sensitive. To enforce uniqueness, you may want to compare them in a case insensitive way.

However, you may wish to also capture the specific casing entered by the user.

How can you accomplish both of those contradictory goals at once?

Significant Casing And Unique Constraints

How might you handle a scenario where you accept and store custom case information, but wish to compare values in a normalized form?

First, enable the citext extension in the Rideshare database. Enter this SQL statement into psql:

```
CREATE EXTENSION IF NOT EXISTS citext;
```

The PostgreSQL module citext¹³ provides a case insensitive character type that can be used for this purpose. Review the installed extensions by running \dx from psql.

13. <https://www.postgresql.org/docs/current/citext.html>

To test this out, create a new table in the Rideshare database but create it in a separate temp schema. Use the temp schema for tables that aren't part of the application schema.

From psql, enter this statement to create Schema if it doesn't exist:

```
CREATE SCHEMA IF NOT EXISTS temp;
```

Alternatively, use the following Migration to enable the citext extension and create a customers table in the temp schema. The email column will use the citext type.

```
ruby/migration_add_customers_table.rb
class AddCustomersTable < ActiveRecord::Migration[7.0]
  def change
    enable_extension 'citext' unless extension_enabled?('citext')

    # Assumes a `temp` schema exists
    create_table 'temp.customers' do |t|
      t.citext :email
    end
  end
end
```

Now that you've added the temp.customers table, insert some data into it. Run the following statement from psql.

```
INSERT INTO temp.customers (email) VALUES ('Customer1@example.com');
```

Try querying for the row you just inserted, but use a lower case "c" in the WHERE clause.

```
SELECT * FROM temp.customers WHERE email = 'customer1@example.com';
```

Despite using the different casing, your query result should include the row. This was made possible using the citext type. What about the Unique constraint portion?

For citext columns, Unique constraints enforce unique values regardless of their casing. Add a Unique constraint and try to insert a row with an email address that has a different casing, and you should see that it can't be inserted.

This means you've stored the original casing, and are also preventing duplicates without needing to also store a normalized value.

What about situations where you do wish to store a normalized value?

Storing Transformations In Generated Columns

PostgreSQL 12 added a feature called *Generated Columns* and it's supported by Active Record as virtual columns.¹⁴ A virtual column is a second column you'll create that receives a transformed value.

You'll try this out to see how it works.

Create a column called `email_downcased`. The following Migration fragment creates the table from scratch with the original `email` column and the `email_downcased` column.

The `email_downcased` column uses the SQL `LOWER()` function to modify the value from `email`.

```
ruby/migration_snippet_virtual_column.rb
create_table 'temp.customers' do |t|
  t.string :email
  t.virtual :email_downcased,
    type: :string,
    as: 'LOWER(email)',
    stored: true
end
```

After adding this, the `email_downcased` has the following Default configuration:

```
GENERATED ALWAYS AS (LOWER(email::text)) STORED
```

The `GENERATED` keyword refers to the Generated Column. `ALWAYS AS ... STORED` means the transformed value is automatically stored that way. The `LOWER()` function changes the `email` column value to lowercase.

Being a native feature, the Generated Column approach does not require adding a new extension like `citext`.

On the other hand, it does require adding a new column to your database.

In the next section, you'll continue exploring ways to improve the consistency and correctness of your data.

Another useful capability are PostgreSQL Enumerated Types. If you're a seasoned Rails developer, you may be familiar with the `Enum` type and be using it in your application code. PostgreSQL supports Enum types stored within the database.

How does that work?

14. https://guides.rubyonrails.org/active_record_postgresql.html#generated-columns

Constraining Values With Database Enums

Enumerated Types or *Enums* are a way to express a limited set of possible values. In a database context, they are similar to Constraints in that they *constrain* the possible values. Enums serve as a form of documentation for what the possible column values are.

When adding an Enumerated Type to your database, it becomes part of your database structure.

In Active Record, an Array of Enum values can be used with an inclusion validation to ensure input data matches one of the types.

Let's look at an example. Although not an Enumerated Type specifically, the `VehicleStatus` class in the Rideshare codebase limits status values to `DRAFT` or `PUBLISHED` Strings.

Rideshare uses an Inclusion validation in the `Vehicle` model to enforce this.

```
validates :status, inclusion: { in: VehicleStatus::VALID_STATUSES }
```

The `status` column also has a `NOT NULL` constraint and a column default of `VehicleStatus::DRAFT`.

How might you implement a similar configuration using PostgreSQL Enumerated Types?

Start with the Rails Guides page covering Active Record and PostgreSQL Enumerated Types.¹⁵ Review the following details.

- Enums are a PostgreSQL type
- When adding Enum values, they can be added before or after existing values, but not in the middle
- Enum values cannot be dropped

Enums look useful, but these limitations are significant and worth considering for your use case.

If you require more flexibility, you may wish to leave the enumerated type in application code. However, if your enumerated type values aren't changing, then the extra rigidity in the database provides a stronger consistency guarantee.

15. https://guides.rubyonrails.org/active_record_postgresql.html#enumerated-types

To get started with Active Record, use the `create_enum` Ruby method. The column is added and the type is set using the enum name. The default value and null handling are the same.

Copy and paste the following code into the `change` method in a generated Migration, and run `bin/rails db:migrate` to apply it.

ruby/migration_snippet_vehicle_enum_usage.rb

```
create_enum :vehicle_status, [
  VehicleStatus::DRAFT,
  VehicleStatus::PUBLISHED
]

add_column :vehicles, :status, :enum,
  enum_type: :vehicle_status,
  default: VehicleStatus::DRAFT,
  null: false
```

After migrating, you've now created a new enum database object. Enums are dumped into the `db/structure.sql` file. You'll see a `CREATE TYPE` statement that looks like this:

sql/create_type_as_enum.sql

```
CREATE TYPE vehicle_status AS ENUM (
  'draft',
  'published'
);
```

If you'd like to explore this further within PostgreSQL, use the `pg_type` system catalog, which may be queried to list Enums. Confirm the enum name and values which should match the Migration above.

sql/view_pg_types.sql

```
SELECT
  n.nspname AS enum_schema,
  t.typname AS enum_name,
  e.enumlabel AS enum_value
FROM pg_type t
JOIN pg_enum e ON t.oid = e.enumtypid
JOIN pg_catalog.pg_namespace n ON n.oid = t.typnamespace;

-- enum_schema | enum_name | enum_value
-- -----+-----+-----
-- public      | vehicle_status | published
-- public      | vehicle_status | draft
```

You may run into a compatibility problem between the database enumerated type and the Active Record model methods. Update the model code to include the enum information.

```

ruby/active_record_enum_usage.rb
enum status: {
  draft: "draft",
  published: "published"
}, _prefix: true

```

The `_prefix` option adds a `status_` prefix to the generated model methods. With that code in place, try calling `status_draft?` on a `Vehicle Reservation` instance. You can instantiate model instances and call methods on them from `bin/rails console`.

To see all of the possible values for the enum, refer to the `db/structure.sql` file.

A benefit of using an Enumerated Type as a column type is that it can be reused with other tables.

The post *PostgreSQL: ENUM is no Silver Bullet*,¹⁶ lists examples of common enumerated types and the trade-offs with Enums in general.

The post *Enums vs Check Constraints in Postgres*,¹⁷ advocates for skipping Enums entirely, and using CHECK constraints with an explicit list of values instead.

You'll have to determine what makes the most sense for your needs, but you've got a variety of options to choose from.

PostgreSQL has even more capabilities for constraining values. In the next section, you'll look Domains which a new object type with similarities to both Enums and Check Constraints.

Read on to see what they offer.

Sharing Domains Between Tables

PostgreSQL Domains¹⁸ are another object type with similarities to Enums and Constraints. The post *Enforcing a Set of Values*,¹⁹ explores storing Enums and Domains.

The Domain created in the post has the type `Text` and uses a CHECK constraint for enforcement in a limited set of values. The Domain name is then used as the column type on a table.

16. <https://medium.com/swlh/postgresql-3-ways-to-replace-enum-305861e089bc>

17. <https://www.crunchydata.com/blog/enums-vs-check-constraints-in-postgres>

18. <https://www.postgresql.org/docs/current/sql-createdomain.html>

19. <https://justatheory.com/2010/01/enforce-set-of-postgres-values/>

From psql, try creating a Domain by copying and pasting the following SQL statement.

```
sql/create_domain.sql
```

```
CREATE DOMAIN vehicle_statuses AS TEXT
CONSTRAINT valid_vehicle_statuses CHECK (
    VALUE IN ('draft', 'published')
);
```

After running the statement, type \dD to “describe Domains” and confirm the Domain you just added shows up as a result. You’ve now created a freestanding vehicle_statuses Domain that’s not yet in use.

Use the vehicle_statuses Domain for the status column in the Rideshare Vehicles table. To do that, run the following SQL. The first statement drops the existing status column so that it can be added back with the domain.

```
sql/vehicle_status_domain.sql
```

```
-- drop column so it can be added back
ALTER TABLE vehicles DROP COLUMN status;

-- add column with a Domain called "vehicle_statuses"
ALTER TABLE vehicles
ADD COLUMN status vehicle_statuses
NOT NULL
DEFAULT 'draft';
```

Once you’ve run this, the vehicles.status column now uses a Domain called vehicle_statuses. The column does not allow NULL values and has a default value.

Row modifications must pass the CHECK constraint definition that is part of the Domain.

The main difference compared with the Enum from earlier is that the NOT NULL constraint is part of the Domain. Enums do not have a way to attach additional constraint types to them.

Like Enums, this Domain can be used by different tables that have the same needs.

You’ve now implemented a variety of mechanisms to help ensure that data is correct, consistent, and referentially intact. These techniques can be used to help you prevent data bugs in your application. With fewer bugs, your team’s development velocity won’t be slowed down due as much for bug fixing.

To help grow the knowledge around these capabilities on your team, you may wish to add some tooling. In the next section, you’ll explore some open source tools in this area.

Automating Consistency Checks In Development

In this section you'll explore some open source tools to help you identify invalid or inconsistent data.

Rideshare includes the `active_record_doctor`²⁰ gem, which can be used to analyze the database and application code.

Active Record Doctor is a command line program. Run `bin/rake active_record_doctor` from the Rideshare directory in your Terminal, which runs built-in checks and makes recommendations.

One of the checks is comparing the column types between Primary Key and Foreign Key columns.

Active Record Doctor identifies mismatched types and recommends changing the types so that they match.

Validations defined in models are also compared against database constraints. For example, when a uniqueness Validation exists but there's no UNIQUE constraint for the same column, the tool recommends adding a UNIQUE constraint. The tool also recommends adding missing Foreign Key constraints.

Another gem to try out is `database_consistency`.²¹ Database Consistency makes also analyzes Active Record models and database objects and makes recommendations.

For example, when a LIMIT is specified on a column in the database, but a similar length limit doesn't exist as a Active Record Validation, a recommendation to add the Validation is offered.

Using Active Record Doctor and Database Consistency as part of your development process can help you identify data quality improvements early.

Now that you've built a solid foundation for data quality, it's time to shift gears more towards the database structure.

Over time you'll add or update your structure, as new designs or more optimal designs come along. For large and busy databases, some types of structural changes are dangerous in that they can block queries and cause partial downtime.

How can you safely make changes despite heavy growth in data and query volume?

20. https://github.com/gregnavis/active_record_doctor

21. https://github.com/djezzl/database_consistency

Part III

Operate and Grow

Modifying Busy Databases Without Downtime

Over time application codebase and database structures evolve. Part of your development team's velocity hinges on being able to have a short time to delivery for both new code changes and database modifications. Rails developers use Active Record Migrations to evolve the database structure, constantly creating DDL changes that are part of feature development. Active Record is used as a change management mechanism and keeps all databases in sync from local development to production.

When your application is new and usage is low, modifying the database structure has low or no risk. Tables that are locked for changes have very short lock durations. There may not be much concurrent activity from customers or internal users using the table at the same time as it's being changed.

As an application grows in popularity and usage, the growth rate and query volume increases, and this can cause database changes that would have otherwise been fun, to error. Time that your development team spends debugging database changes negatively affects your team development velocity.

One of the main challenges you'll face is continuing to evolve your database as needed despite huge growth.

How do you identify problematic database structure changes and avoid some of the pitfalls? You'll learn more about that in this chapter.

Besides changes to your database structure, you'll also conduct large scale *backfills*. Backfills populate one or more columns that didn't exist before, and the population of large amounts of rows carries some risk.

Backfilling on tables with millions or billions of rows presents some technical challenges you'll look at later in this chapter.

Take a look at some of the terminology you'll be using.

Busy Databases Terminology



- Strong Migrations — Library that adds more safety to Migrations
- Multiversion Concurrency Control (MVCC) — PostgreSQL mechanism for managing row modifications and concurrent data access
- ACID — Guarantees PostgreSQL makes including Atomicity, Consistency, Isolation, Durability
- Isolation Levels — Configurable access level for transactions
- Denormalization — Duplicating some column data
- Backfilling — Populating new columns for large amounts of rows based on a new design
- Table Rewrites — Internal changes from schema modifications that cause significant availability delays

Some changes are dangerous at a higher scale. How can you find those?

Identifying Dangerous Migrations

As an experienced Rails developer and PostgreSQL user, you've likely had Migrations that didn't run correctly. These might have resulted in failed deployments that blocked your release process. You'd like to detect these earlier and avoid as many failed migrations as possible.

One solution would be to take your database down and perform changes offline. With a disconnected database there would be no concurrent modifications to worry about. Structural changes would be perfectly safe!

Unfortunately that strategy is impractical. Modern development teams don't take their databases offline to make structural changes. Modern teams continually ship code changes and schema evolutions every day. Fortunately PostgreSQL can keep pace with the needs of your team, but you'll need to add new processes around detection and action steps when dangerous migrations are detected.

Active Record Migrations have no built-in concept of safety or danger. All migrations are treated as being equally safe or unsafe.

Fortunately, third party gems like Strong Migrations¹ add this useful information and more. Strong Migrations identifies potentially unsafe migrations during the development process, by comparing the Migration content against well known problematic patterns. Since it's hooked into the normal Migrations process, it can even prevent unsafe migrations from running at all. To help developers out, Strong Migrations provides safer alternatives when an unsafe Migration is detected.

Learning From Unsafe Migrations

To get some hands-on experience with the effect of blocked queries, you'll deliberately create one. The unsafe Migration you'll work with is called adding a Volatile default.

To do that, you'll create a long running modification that adds the default value, and then run a query in that time that will be blocked. You'll use Rideshare and the temp schema from earlier.

Launch psql from your terminal, connecting to the Rideshare database. To reset the temp schema, use the CASCADE option which drops any tables in that Schema. Then create it again.

```
DROP SCHEMA IF EXISTS temp CASCADE;
CREATE SCHEMA temp;
```

Within the temp schema, create a slimmed down users table with an id and name column.

Populate the table with 10 million rows. Use the CREATE TABLE AS form of creation to create the table and populate it in a single statement.

Use GENERATE_SERIES() to get Integer values from 1 to 10 million for the id and to create a unique name.

Toggle timing with \timing to see how long the operation takes.

Run the following statements in psql. This will take a bit of time to populate.

```
sql/migration_dangerous_defaults_setup.sql
-- Enable timing
\timing

-- create users table (id, name)
```

1. https://github.com/ankane/strong_migrations

```
-- populate it with 10,000,000 rows
CREATE TABLE temp.users AS
SELECT
    seq AS id,
    'Name-' || seq::TEXT AS name
FROM GENERATE_SERIES(1, 10000000) AS t(seq);
```

Verify the temp.users table structure by running `\d temp.users`. Confirm it has 10 million rows.

Imagine you've made an application change and want to associate Users to Cities. You'll track a City for the user in a new `city_id` column. You want both new and existing Users to have a City assigned to them.

In the application, you might prompt users to add their City, but you'd like the column to have a default value to start.

Add a new `city_id` column and give it a default value.

You'll perform this two times to illustrate a difference between how a Volatile and Non-volatile Default value works. The first one is relatively safe since it uses a Non-volatile Default. The second version is unsafe on a large table because of the Volatile Default.

Run the statement below which adds a Non-volatile "static" value of 1 to all rows.

```
sql/migration_safe_modification.sql
-- add column with a constant default value
-- This is safe to do and runs fairly quick
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
    DEFAULT 1;
```

After adding the column, run `\d temp.users` and make sure `city_id` has the Default defined.

Drop the column you just added to prepare for adding it a second time.

```
ALTER TABLE temp.users DROP COLUMN city_id;
```

For the next run, it will help to have two different sessions. The examples below have `psql1` and `psql2` in their name referring to where you should run the statements.

In the unsafe version, you'll run the ALTER TABLE statement from the `psql1` session.

Run that now.

```
sql/migration_dangerous_modification_psql1.sql
-- !!! Dangerous Version !!!
-- Adds a "non-constant" or "volatile" DEFAULT
-- This takes a LOT longer
-- Table is locked in 'ACCESS EXCLUSIVE' mode for duration
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
    DEFAULT 1 + FLOOR(RANDOM() * 25);
```

While that's running, from the psql2 session, run the following quick and simple SQL query:

```
sql/migration_dangerous_modification_psql2.sql
SELECT * FROM temp.users LIMIT 1;
```

The DDL in psql1 might take ten seconds or more to run. In that time, the quick and simple query in psql2 will appear “hung” until the modification in psql1 finishes. Once psql1 finishes, psql2 runs instantly.

psql2 appears hung because psql1 has locked the table with exclusive access.

Now imagine instead of the single query for psql2, there are hundreds of client application queries trying to run during that lock period, getting blocked. This could quickly cause thousands of errors for your application. This is something you'll want to avoid!

What steps can be put in place to help detect and avoid this scenario?

Learning To Use CONCURRENTLY By Default

PostgreSQL provides the CONCURRENTLY keyword as an option for a number of operations. Learn to use the CONCURRENTLY keyword whenever it's available as the default way that you work. Note that CONCURRENTLY is *NOT* added by default, so it will be up to you to specify. When an table has no other concurrent activity, using CONCURRENTLY won't provide a benefit.

Let's start with some of the places where you'll use it. You haven't yet learned about Rebuilding Indexes or managing Partitions. For now you don't need an understanding of what those are, only that they support the CONCURRENTLY keyword.

- Adding an Index
- Dropping an Index
- Rebuilding an Index
- Adding a Partition
- Detaching a Partition

Using `CONCURRENTLY` can make an operation like Creating an Index take twice as long. You saw in an earlier Chapter when building the Performance Database, for offline modifications using `CONCURRENTLY` is not needed. In those cases you wanted to prioritize the speed of an operation like building an Index, vs. handling concurrent activity safely.

Using `CONCURRENTLY` is a small part of your safety checks and processes. What other safety mechanisms can you add?

Adopting A Migration Safety Check Process

Each Active Record Migration file is a new source code file. At organizations that use Pull Requests or Merge Requests to introduce changes, a single change is prepared and a developer reviews it.

Experienced developers might identify dangerous database changes from mistakes they'd made in the past. Raising these concerns in a code review can help prevent them from getting released.

While reviewing feedback from team members is valuable and may even be required, it does imply that team members have the necessary skills to distinguish safe from unsafe migrations, and that they're available to perform reviews.

Tooling can help augment this process, identifying potentially unsafe Migrations like an experienced developer might do. Ideally the tool was available to all developers on the team during their local development process so they'd get feedback early.

Fortunately this tool exists and it's called Strong Migrations!

Exploring Strong Migrations Features

Strong Migrations strengthens the uptime and resiliency of Rails apps and engineering teams by detecting unsafe Migrations and by providing and requiring safer alternatives.

Most unsafe operations are detected and prevented by default. Besides the stability benefits, there is also an educational benefit by integrating the knowledge around the known problematic patterns and solutions into the development process. The Strong Migrations GitHub README includes a wealth of information.

Some of the operations that are detected are listed below.

1. When adding constraints (CHECK, FOREIGN KEY, and NOT NULL) to an existing column, Strong Migrations checks whether they're validated immediately or not.
2. Strong Migrations checks for table modifications *and* data backfills that would extend the duration of an exclusive lock.
3. When removing or renaming a column or table, Strong Migrations checks whether this will create a problem with the Schema Cache.
4. Strong Migrations prevents changing the type of a column to an incompatible type.
5. When Adding or Dropping an Index without the CONCURRENTLY keyword, Strong Migrations prompts the developer to use CONCURRENTLY.

How does it work?

Strong Migrations hooks into the Rails Migration process when developers run `bin/rails db:migrate`.

When a potentially unsafe migration is detected, a Ruby Exception is raised preventing the migration from being applied.

This forces the developer to either abandon it, rewrite it using a safer alternative, or override the safety check with the `safety_assured` Ruby block. Review the `db/migrate` directory of Rideshare for some examples.

Strong Migrations has a good track record of catching unsafe migrations.

Next you'll look at a specific example from Rideshare.

Trips can have a rating of Integer values between 1 and 5. As an Active Record Validation, Rideshare uses a “numericality” validator configured in the Trip model to enforce this. The rating value can also be nil since trips aren't rated until after they're completed.

Review the rating field validation in `app/models/trip.rb`.

The rating validation can also be implemented in PostgreSQL using a CHECK constraint.² A CHECK constraint for this purpose was added to Rideshare. Review the following Migration file that added it.

`ruby/migration_trip_rating_check_constraint.rb`

```
class AddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :users,
      "rating >= 1 AND rating <= 5",
      name: "rating_check"
```

2. <https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS>

```
end
end
```

The Migration used the `add_check_constraint` method added in Rails 7. This Migration was flagged by Strong Migrations as unsafe. Initially it did not use the `validate: false` option, meaning that it would be applied to all rows of the table.

The solution involves two steps. First, add `validate: false` to the Migration, which means it will in effect only for new Inserts, Updates, and Deletes.

Second, generate a new Migration file that only Validates the Check Constraint by name. These two Migrations are combined into a single code example below.

```
sh/migration_check_constraint_dangerous.rb
# Example message from Strong Migrations:
#
# === Dangerous operation detected #strong_migrations ===
#
# Adding a check constraint key blocks
# reads and writes while every row is checked.
# Instead, add the check constraint without
# validating existing rows, then validate them
# in a separate migration.
#
# Migration #1: Add Check Constraint using "validate: false"
# Add a migration with "validate: false"
class AddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :trips,
      "rating >= 1 AND rating <= 5",
      name: "rating_check",
      validate: false
  end
end

# Migration #2 (generate new file): Validate Check Constraint
# using `validate_check_constraint` method
class ValidateAddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    validate_check_constraint :trips,
      name: "rating_check"
  end
end
```

Adding a CHECK constraint to a table requires an Exclusive Access lock while being added. By using the `validate: false` option, the exclusive lock period adding the Constraint will be extremely brief, causing little or no interruption.

To enforce consistency for existing rows, you'll need to Validate the constraint. This used the second Migration which called the `validate_check_constraint` method.

Before the constraint was validated, running `\dt trips from psql` would show the Constraint with the NOT VALID keywords:

Check constraints:

```
"rating_check" CHECK (rating >= 1 AND rating <= 5) NOT VALID
```

Validating the constraint will remove the NOT VALID keywords. If there are Constraint violations, you'll learn about them at that time, and as you saw earlier, you'll need to fix them.

With a validated constraint, you are now confident that all rows are consistent with the CHECK constraint definition. You were able to add the Check constraint even for a large and busy table without causing downtime.

Now that you've started to focus more on locks, blocks, and concurrency, let's refresh some PostgreSQL knowledge on these topics in the next section.

Locking, Blocking, and Concurrency Refresher

As you've learned earlier, one of the operational challenges with PostgreSQL is minimizing disruptive table locks when modifying the table structure. This section will go into more depth in that area.

All SQL statements, including SELECT statements (DQL), DML, and DDL, run inside a transaction.

Transactions are normally automatic and implicit, although as the developer you may use explicit transaction blocks with the keywords BEGIN and then COMMIT or ROLLBACK. You may also lock resources explicitly.

To further confuse the matter, the Active Record `disable_ddl_transaction!` is problematic. See *PR #21044*³ which explains how Disabling the wrapping transaction is not limited to DDL, but could be for a DML operation. Other databases don't support Transactional DDL. Even with this option set, a Rails client application can't change the PostgreSQL Transactional DDL feature, meaning DDL still runs inside an implicit transaction.

What about lock types? PostgreSQL reports which lock types conflict with each other in the *Conflicting Lock Modes* table on the *Explicit Locking*⁴ documentation page. Familiarize yourself with different DDL and DML statement

3. <https://github.com/rails/rails/issues/21044>

4. <https://www.postgresql.org/docs/current/explicit-locking.html>

types and which locks they take, or jump straight into PostgreSQL Locks Conflicts.⁵

If you do have lock contention issues, how can you gain more insight into what's happening?

Another System Catalog you can use is the `pg_locks` view. Review the *Lock Monitoring*⁶ PostgreSQL wiki for more information.

Lock contention is not something you'll usually see and fix in real time, so it can be difficult to have a mental model of it. One way to build that up is to simulate a lock scenario. You'll do that next!

In this exercise you'll create two transactions that contend with one another.

Open two psql sessions, named `psql1` and `psql2` as you did in an earlier section.

```
sql/locks_and_blocks_part_1_psql1.sql
```

```
-- From "psql1" connected to Rideshare:
-- psql --dbname rideshare_development
-- Start a Transaction
BEGIN;

-- create explicit EXCLUSIVE LOCK of trips table
LOCK trips IN ACCESS EXCLUSIVE MODE;
```

```
sql/locks_and_blocks_part_2_psql2.sql
```

```
-- From "psql2"
-- Inspect pg_locks view
-- Find the "AccessExclusiveLock" for "trips"
SELECT
    mode,
    pg_class.relname,
    locktype,
    relation
FROM pg_locks
JOIN pg_class
ON pg_locks.relation = pg_class.oid
AND pg_locks.mode = 'AccessExclusiveLock';

-- -[ RECORD 1 ]-----
-- mode      | AccessExclusiveLock
-- relname    | trips
-- locktype   | relation
-- relation   | 461492

-- "psql1", inspect the current lock_timeout;
-- A lock_timeout of 0 means it is disabled, so
```

5. <https://postgres-locks.husseinnasser.com>

6. https://wiki.postgresql.org/wiki/Lock_Monitoring

```

-- a statement waiting on a lock will run forever
SHOW lock_timeout;

-- "psql2"
-- Create the DDL statement from earlier.
-- DDL statement acquires an exclusive lock
-- But it will get blocked!
ALTER TABLE trips ADD COLUMN city_id INTEGER DEFAULT 1;

-- Since this hanging go back to "psql1"

```

```

sql/locks_and_blocks_part_3_psql1.sql
-- Query pg_stat_activity from "psql1"
-- Look for waiting transactions
-- Confirm the ALTER TABLE is listed and blocked
SELECT
    wait_event_type,
    wait_event, query
FROM pg_stat_activity
WHERE wait_event = 'relation'
AND query LIKE '%ALTER TABLE%';

-- -[ RECORD 1 ]-----+-----
-- wait_event_type | Lock
-- wait_event      | relation
-- query           | ALTER TABLE trips ADD COLUMN city_id INTEGER DEFAULT 1;

-- NOTE: UNBLOCK
-- To unblock, COMMIT or ROLLBACK the original transaction from "psql1"
ROLLBACK;

```

The ALTER TABLE statement gets blocked trying to modify the trips table because the trip table is locked with exclusive access. The statement is waiting to acquire the same lock type.

How might you prevent the statement waiting forever?

Preventing Excessive Queueing With A Lock Timeout

One protection you should add is a lock_timeout. The purpose of the lock timeout is to set a maximum allowed time a transaction can wait before it's canceled.

The lock_timeout⁷ parameter can be set database wide, within a session, or for a single SQL statement.

In the example from the last section, the lock_timeout was not set.

Try running the experiment again but this time set a lock_timeout. You'll need two psql sessions psql1 and psql2 as you did earlier.

7. https://postgresqlco.nf/doc/en/param/lock_timeout/

Use the statements below. The main difference from earlier is that you're using `SET LOCAL lock_timeout = '5s';` to set a `lock_timeout` of 5 seconds within a transaction.

```
sql/set_lock_timeout_psql1.sql
```

```
-- create psql1
-- start a transaction
BEGIN;
LOCK trips IN ACCESS EXCLUSIVE MODE;
```

```
sql/set_lock_timeout_psql2.sql
```

```
-- create psql1
-- start a transaction
BEGIN;
LOCK trips IN ACCESS EXCLUSIVE MODE;

-- create psql2
-- set a transaction level lock_timeout
BEGIN;
SET LOCAL lock_timeout = '5s';

-- Run the modification
-- It should hang since the table is locked for exclusive access
-- But it should get canceled after 5s
ALTER TABLE trips ADD COLUMN city_id INTEGER;

-- In psql2 notice the statement is canceled
-- ERROR: canceling statement due to lock timeout

-- run rollback in both psql1 and psql2
ROLLBACK;
```

With the timeout in place, the statement will be canceled after waiting the maximum amount of time. This adds a safeguard that prevents your transactions from waiting forever.

What other kinds of safeguards can be added?

Exploring Lock Type Queues

Statements requesting access for a lock type are put into a queue where they wait in order.

If transaction B is waiting to acquire the lock type that transaction A holds, B must wait for A.

As described in *PostgreSQL and the Lock Queue*,⁸ the pids for the backend processes are logged including the statement holding a lock type and the statements waiting to acquire it.

8. <https://joinhandshake.com/blog/our-team/postgresql-and-lock-queue/>

Some helpful lock parameters to set are `log_lock_waits` to on and setting a value for `deadlock_timeout` to gain visibility into locks or blocked queries. When these timeouts cause cancellations, they'll be logged to the `postgresql.log`. See the post *Cost of 'log_lock_waits' and 'deadlock_timeout'*.⁹

Take a look at the following example. This example shows the pid holding a lock and the pids waiting for the lock type.

```
DETAIL: Processes holding the lock: 29386. Wait queue: 31025, 30551, ...
```

The pids 31025 and 30551 are blocked waiting on pid 29386. Once the query associated to the backend pid 29386 completes, the pids that are waiting will be processed in the order of how they appear.

Besides using the `pg_stat_activity` system catalog you learned about earlier, you may want to use a log analyzer that analyzes your `postgresql.log` file.

The log analyzer `pgBadger`¹⁰ organizes lock related query information it parses from your log file. The lock information is put into categories like “Most frequent waiting queries” and “Queries that waited the most” which can help your investigations.

Next you'll look at the Statement Timeout.

Setting Statement Timeout

The Statement Timeout can be set from the client application to set a maximum allowed time for statements. When the time is reached, the statements are cancelled. A Statement Timeout has been set in Rideshare in the `config/database.yml`.

When queries are canceled for taking too long, they'll need to be fixed so that they run in time. This is especially crucial for user-facing Rails queries, although background job queries should be sped up as well.

Setting too low of a value for `statement_timeout` cancels queries unnecessarily. Not setting it at all means queries are allowed to run forever, consuming limited resources like database connections. A happy middle ground is to set a reasonable `statement_timeout` value that balances these two extremes.

In a Ruby on Rails web application, user-facing queries should run in less than 250ms. Ecommerce application queries may need to run even faster, in less than 50ms. B2B Enterprise SaaS applications may tolerate slower queries.

9. <https://dba.stackexchange.com/a/249904>

10. <https://github.com/darold/pgbadger>

In the long run, fast queries make it easier for the same server to achieve higher levels of concurrency, since the queries are more efficient. This efficiency can prevent or delay the need to vertically scale the database instance.

If your application does not have a statement timeout set at all, try starting with a high value like 10 or 15 minutes. As you fix queries and some time passes without seeing cancellations, lower the timeout value. Repeat this pattern and lower the value further.

You may want to make some exceptions. Allocate a small portion of the total database connections to serving slow queries. You may also want to allow a higher timeout value for background job queries.

This concludes timeout values. In the next section you'll shift gears and look at how Active Record caches your schema.

Avoiding Schema Cache Errors

When Active Record starts up, for each model that's backed by a database table, the table fields are scanned and cached in the *Schema Cache*¹¹ for as long as the application is running.

Dropping a database column without considering Active Record and active instances of the application means that a schema change may cause an application error. This is due to the application trying to write or read to a cached column that's been removed.

To drop a column safely, use the Active Record `ignored_columns`¹² mechanism to add the field you're planning to drop first. Add the field and deploy that change, allowing the application servers to restart.

A field added to `ignored_columns` can safely be dropped from the database without concern that there are any references to it that are stale in the Schema Cache.

You're now ready to consider Backfilling and some of the operational challenges associated with it.

Backfilling Large Tables Without Downtime

Over time your application structure will evolve. You may introduce a new column that gets populated for new rows, but is NULL for the rows that existed before the feature was added.

11. <https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaCache.html>

12. <https://www.bigbinary.com/blog/rails-5-adds-active-record-ignored-columns>

Populating old rows is usually called *backfilling*. As your table row counts grow large, backfills take longer and can become so complicated and risky that you may not want to do them.

This can be acceptable in some cases. Backfills could be optional if the application can fall back to a default value. When that's not possible, you'll need to backfill.

While you may conduct backfills from a Ruby application using a Rake task, in this section you'll conduct backfills using SQL and `psql`.

Backfills should not harm client application queries that are running at the same time. To accomplish that, your backfill operations need to run efficiently and consume few resources. Ideally your server instance is over-provisioned as well, meaning it has a lot of spare compute and disk IO capacity.

From a timing perspective, run backfill queries during a low activity period. Capture the SQL being run in a SQL file or script, share it on your team inviting feedback, and validate the changes in a pre-production database first.

Some backfilling concepts you'll see are:

- Using a *Double Writing* or “Dual Writes” strategy
- Designing an intermediate table to assist in the backfill
- Using batched operations to limit resource consumption
- Adding a throttling mechanism to slow down operations if needed. This allows for Index Maintenance and Replication.
- Creating specialized indexes based backfill queries.
- Making the table Unlogged and disabling Autovacuum

In the next sections you'll explore some of these techniques in more detail.

Backfilling And Double Writing

Modern engineering organizations are able to perform data migrations online for most cases, avoiding the need for downtime. Downtime halts business operations and should be avoided if possible.

Zero or minimal downtime approaches can be challenging technically and risky compared with an offline backfill. Online backfills typically require careful planning and execution.

After introducing a new database column, you may want to plan out multiple phases where first the write operations are cut over, and then the reads.

This technique can be called *Double Writing*. Double writing can help as a transitional technique to minimize the time needed in a cutover where reads are switched from the original location to a target location, by ensuring the data population is conducted for both locations at the same time.

Separating Reads and Writes for Backfills

Backfills are straightforward when they can `SELECT` from one table and `UPDATE` another table.

When the table being queried is under heavy load, it may be too slow to query it in high volume for backfill purposes. Determine if you can modify the table with special-purpose indexes to speed up the backfill query.

If the queries to read and write aren't fast enough, you may want a special-built table as a transitional element.

Specialized Tables For Backfills

When optimizing the read side and write side queries is not enough, you may need an intermediate table to assist in the backfill process.

The data in this table will duplicated from a source. There will be no other connections to or contention for this table since it will only exist for the backfill. This separate table can also be used to track the backfill progress.

A downside of this approach is that since you're duplicating data, you'll consume more space.

Since this table is designed specifically for the backfill, you'll add the most narrow set of columns possible, and custom indexes to it to help with backfill queries.

To make writes to the table even faster and avoid having this table replicated, set the table to `UNLOGGED`.

From `psql`, run the `SET UNLOGGED` statement for the table. An unlogged table does not have write protection from the Write-Ahead Log (WAL). If PostgreSQL restarts unexpectedly, you may lose data for this table. Since this table is a secondary duplicate though, the impact from it being unlogged is near zero because the table can be re-created if needed.

Autovacuum can also be disabled for a transitional intermediate table. If the table was called `trip_requests_intermediate` then you'd run this SQL statement to disable Autovacuum:

```
sql/alter_table_disable_auto_vacuum.sql
```

```
ALTER TABLE trip_requests_intermediate
SET (autovacuum_enabled = false);
```

Next you'll put intermediate table techniques into practice with Rideshare.

Practicing Backfilling Techniques

Let's practice some of the backfill techniques to develop your skills in this area.

Using the example where you added a `city_id` column to the `users` table, imagine that you were creating a purpose-built intermediate table that would act as the data source to select from.

This table has these characteristics:

- A minimal set of columns
- Purpose made index for a single query
- No constraints
- Unlogged
- Autovacuum disabled

Run these statements in `psql`:

```
sql/create_backfill_intermediate_table.sql
```

```
CREATE SCHEMA IF NOT EXISTS temp;

CREATE UNLOGGED TABLE temp.users_intermediate (
  user_id BIGINT, city_id INTEGER
);

ALTER TABLE temp.users_intermediate
SET (autovacuum_enabled = false);
```

Generate some rows:

```
sql/backfill_intermediate_generate_rows.sql
```

```
INSERT INTO temp.users_intermediate (user_id, city_id)
SELECT GENERATE_SERIES(1, 10, 1), GENERATE_SERIES(1, 10, 1);
```

For your backfill, you'll `UPDATE` the `users.city_id` column, using matching rows from the `temp.users` table. If the `users.city_id` column does not exist, you can add it by running this statement in `psql`:

```
ALTER TABLE users ADD COLUMN IF NOT EXISTS city_id INTEGER;
```

Imagine that `temp.users_intermediate` had millions of rows. You want to select rows as fast as possible.

To do that, add a Multicolumn index covering both columns:

```
CREATE INDEX temp_users_user_id_city_id
ON temp.users_intermediate (user_id, city_id);
```

You'll learn more about Multicolumn indexes in an upcoming chapter. Since you have few rows in the table, PostgreSQL will choose a Sequential Scan. Try disabling Sequential Scans for your session by running `SET enable_seqscan = OFF;`.

From psql, run the following query and add `EXPLAIN` in front of it. You haven't started working with `EXPLAIN` and the query planner yet, but they'll be covered more in an upcoming chapter.

The query execution plan output should show the Index name you just created, and an Index Scan should be used.

```
sql/backfill_update.sql
```

```
UPDATE users
SET
    city_id = temp.users_intermediate.city_id
FROM temp.users_intermediate
WHERE users.id = temp.users_intermediate.user_id
AND users.id > 0
AND users.id < 10000;
```

For your batch size, use a range that is sized appropriately for your database server. Iterate through the users table using the primary key id column. There may be deleted rows in the range, but you can plan on there being up to 10,000 rows updated at once for this batch size.

Use the PL/pgSQL Procedure you saw earlier to turn this single statement into a loop that can move through all rows.

Besides batched updates, consider adding pauses within your updates to add time for index writes and replication. Monitor server resource consumption while your statements are running.

Once the destination side table is updated, you can safely remove the temp.users_intermediate table.

Wrapping Up

In this chapter you considered how to safely make database structural changes on large and busy databases. You saw lock types and queues, and learned how queries can become blocked. You simulated lock behavior, blocked queries, and added safeguards by setting a lock timeout and statement timeout.

You learned about backfills and some techniques to perform them faster.

You worked a lot in PostgreSQL. In the next chapter you'll shift gears and work more with Active Record and Ruby code. You'll learn how to optimize queries from application code. Read on to get started.

Optimizing Active Record

Now the focus switches away from PostgreSQL and SQL and to the Active Record query layer in your Rails application.

You'll see a variety of ways to improve the performance of your application by eliminating unnecessary queries, using caches, and using advanced query designs.

Active Record is the *Domain Specific Language* (DSL) that Rails developers use to create SQL queries from Ruby code. Queries are generated and fragments of queries as Ruby code can be composed together.

If you've written SQL before and are newer to Rails, you may wonder whether writing your queries as Active Record is worth it. You may prefer to write SQL directly. What are some of the reasons to prefer Active Record?

Preferring Active Record Over SQL

Active Record has always been the sole built-in ORM framework for Ruby on Rails. Ruby on Rails is one of the most popular open source Ruby projects, meaning many developers learned Ruby primarily in working with the Ruby on Rails framework.

As a full-stack framework, Ruby on Rails has always been opinionated or “prescriptive”, including all of the major layers needed to build web applications, especially an ORM that connects Ruby language objects and instances to database tables and rows.

Rails advocates for *Convention Over Configuration* as a design philosophy. Active Record is the conventional way to describe your application data model and do write queries. Whether a developer is familiar with SQL or not, by embracing the conventional way to write queries, a team may have an easier time growing their contributor base and maintaining their application.

You'll see how to write basic and advanced Active Record queries, as well as write SQL within Active Record. Active Record doesn't limit you to writing SQL queries solely in Ruby, so you've got options.

In Active Record, fragments of SQL queries like filter conditions in a WHERE clause are called *Named Scopes*. The Named Scopes pattern provides a conventional technique to identify most of the “filtering code” by reading through application models. Named Scopes can be shared among models as well and there can be default scopes that get applied to all queries for a particular model.

One example where common filtering occurs and reuse helps is with “soft deletes.” In a conventional soft deletes design, users delete records which are then hidden from the application but not physically removed as rows. Instead they have a `deleted_at` timestamp column that's set. This pattern is implemented in a common location as Ruby code leveraging the Active Record Named Scopes and Default Scope concepts.

Active Record allows a developer to start with a model class and gradually add methods to it. For this chainable design, query generation is deferred until the end, which is called lazy evaluation.

While this design is a good developer experience, it comes with a trade-off: developers can easily write inefficient queries, sometimes unknowingly.

Before you can fix inefficient queries, you'll need to see where they are coming from in the application.

Query Logs To Connect SQL to App Code

Although you may analyze slow queries in an Application Performance Monitoring (APM) tool, when analyzing SQL queries in the `postgresql.log` you can trace them back to the Active Record that generated them.

In this section, you'll learn how to link SQL queries to application code. Prior to Rails version 7, third party gems fulfilled this purpose, but eventually support was added directly to Rails for this very useful functionality.

Prior to Rails version 7, the `Marginalia`¹ gem added comments to SQL queries that showed where in the application the query was generated. The `Marginalia` comment lists the application name, controller name, and controller action name. An example comment looks like this:

```
/*application:Rideshare,controller:trips,action:index*/
```

1. <https://github.com/basecamp/marginalia>

This comment shows that the query was generated from the Trips Controller Index Action in the Rideshare application. The Marginalia documentation advises against adding unique value components when using Prepared Statements because a unique value will prevent re-use of them. You'll learn about Prepared Statements soon.

Since knowing where the query was generated is extremely useful, this capability was added directly to Rails 7 and is called "Query Log Tags."

Rideshare enables Query Log Tags by setting `config.active_record.query_log_tags_enabled` to `true` in `config/application.rb`. Note that Prepared Statements is incompatible with Query Log Tags (See the Warning in the `config.active_record.query_log_tags_enabled` section²).

For this section, Prepared Statements were disabled in the application by putting `prepared_statements = false` in `config/database.yml`.

With Prepared Statements disabled and Query Log Tags enabled, Rideshare will show similar comments to the Marginalia gem in the logged SQL queries. You'll even notice these in the logged SQL in the server log or console log output. For example, notice `/*application:Rideshare*/` comment and run the example below from `bin/rails console`.

```
irb(main):020:0> Trip.first
Trip Load (0.5ms) /*application:Rideshare*/ SELECT "trips".*
FROM "trips" ORDER BY "trips"."id" ASC LIMIT $1 [["LIMIT", 1]]
=>
```

Now you've got a new query observability tool that can help you as you work on improving your application queries.

What are some of the common query problems that can happen?

Common Active Record Problems

The following terminology guide can be used to help learn terms or refresh your knowledge on terms used in this chapter.

Active Record Optimization Terminology



- Normalization — Removing duplication from fields
- Lazy Loading — Loading data "late" and on-demand
- Eager Loading — Loading data "early" or upfront

2. <https://edgeguides.rubyonrails.org/configuring.html#config-active-record-query-log-tags-enabled>

Active Record Optimization Terminology

- **Strict Loading** — Preventing Lazy Loading, requiring Eager Loading
 - **Counter Cache** — Storing a count in a column and incrementing it at insert time
-

While in-depth coverage of data normalization is beyond the scope of this chapter, the basics will be covered here including how they were put into practice for Rideshare.

Good data normalization suggests that your design should eliminate duplication of attributes between tables when possible. This means that each database table has a unique set of fields. When data needs to be linked together, Foreign Key columns refer to Primary Key columns and this relationship is both described and enforced using constraints.

Data relationships might take the form of *one to one*, *one to many*, or *many to many*.

Most Rails applications have many data models and those models are related to each other. Active Record code loads data from many tables, which means rows are joined together in various ways. This is where one of the pitfalls can come in, when rows from multiple tables are not joined together efficiently.

In the earlier section you learned about lazy evaluation of Active Record code, which causes Lazy Loading of data. Lazy Loading means that the data is not loaded upfront. Rails has a different term for loading data upfront: *Eager Loading*.

What's the relevance of Eager Loading? The Eager Loading technique is commonly the solution to one of the most common and well known types of inefficient queries, the *N plus one (N+1)* query pattern. As a refresher on the N+1 pattern, it involves queries for at least two tables, where an outer query runs once, and an inner query runs N times.

Let's break it down further. Querying the right side or inner side repeatedly is a problem because usually it's from Lazy Loading data that could otherwise be Eagerly Loaded.

Let's look at an example from Rideshare. Open `bin/rails` console from your Terminal, connecting to the Rideshare database.

Make sure SQL queries are logged to your console and run this code:

```

ruby/n_plus_one_example.rb
# Show SQL if not already enabled in bin/rails console
ActiveRecord::Base.logger = Logger.new(STDOUT)

Vehicle.all.each do |vehicle|
  vehicle.vehicle_reservations.count
end; nil

```

The count of vehicle_reservations for a Vehicle is queried for every Vehicle.

The SQL queries will look like these:

```

ruby/n_plus_one_queries.rb
Vehicle Load (2.1ms) \
  SELECT "vehicles".* FROM "vehicles" /*application:Rideshare*/
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 1
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 2
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 3
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 4

```

N+1 queries can happen in any Active Record whether it's in the model, views, or serializers. You'll need to discover N+1 instances by monitoring your Rails log or test log, or a team member may spot them during code review.

Manual checks are useful, but automated tools exist that can help you identify N+1 queries as well. The Ruby gem *prosopite*³ is an N+1 detection gem and it's been added to Rideshare. *Prosopite* depends on *pg_query*⁴ for PostgreSQL which was also added.

Let's see if *Prosopite* picks up this N+1. Run the code again in Rideshare in bin/rails console inside of a *Prosopite* block. Set *Prosopite*.rails_logger = true.

```

ruby/n_plus_one_detection.rb
Prosopite.rails_logger = true

Prosopite.scan do
  Vehicle.all.each do |vehicle|
    vehicle.vehicle_reservations.count
  end
end

```

Running this code shows “N+1 queries detected” in the Rails Console. *Prosopite* can even be configured to detect N+1s in tests and fail the tests.

Now that you can detect N+1 queries manually and automatically, how do you fix them?

3. <https://github.com/charkost/prosopite>

4. https://github.com/pganalyze/pg_query

Use Eager Loading To Reduce Queries

To fix N+1 queries, use Eager Loading and eagerly load associated data. You'll need to test to make sure you get the same query results. Usually one query is more efficient even if it takes a little longer, but you'll need to test a bit to confirm.

Active Record Associations (See: *Types of Associations*⁵) like `has_one`, `has_many`, or `belongs_to` to describe different types of relationships and generate different SQL queries to join rows of data from tables together.

Models can be configured for one of six supported types. When you see repeated queries for the same table there's a good chance all of the data can be queried for that table in one query.

Preload data using the `.preload()`⁶ method to combine multiple queries into a single statement.

In bin/rails console, run the same code again but add `.preload()`.

```
Vehicle.preload(:vehicle_reservations).all.each do |vehicle|
  vehicle.vehicle_reservations.size
end; nil
```

This code avoids the N+1 query pattern but it also adds one more trick. `.count()` was replaced with `.size()` which means that when all of the Vehicle Reservations were loaded at once for all of the Vehicles, the size of the loaded collection was used instead of issuing a SQL COUNT() query to count how many there were.⁷

Preloading isn't the only method of Eager Loading data. Read on to learn more.

Eager Loading With `.includes()`

Besides `.preload()`, `includes()` is a second technique to Eager Load data and performs an equivalent SQL query.

Try running the following query with `.includes()` and compare the SQL queries.

```
Vehicle.includes(:vehicle_reservations).each do |vehicle|
  vehicle.vehicle_reservations.size
end; nil
```

5. https://guides.rubyonrails.org/association_basics.html#the-types-of-associations

6. <https://api.rubyonrails.org/classes/ActiveRecord/QueryMethods.html#method-i-preload>

7. <https://www.speedshop.co/2019/01/10/three-activerecord-mistakes.html>

When using `.preload()` the WHERE clause conditions for the second table being joined cannot be modified. The resulting SQL query uses an IN clause for the top level query with a list of primary keys from the second table.

If `.includes()` generates an equivalent SQL query then what value does it add?

With `.includes()`, Active Record can pick between two strategies. `.includes()` will work equivalently to `.preload()` or it will generate a LEFT OUTER JOIN to join the tables together. How does it choose which one to use?

Active Record chooses a LEFT OUTER JOIN when there are conditions on the associated table. What do conditions on the associated table look like?

Modify the example from earlier that fetches Vehicle Reservations, but now fetch the active and not-canceled reservations.

Run the following code in bin/rails console and take a look at the generated SQL query.

```
ruby/n_plus_one_detection.rb
Prosopite.rails_logger = true

Prosopite.scan do
  Vehicle.all.each do |vehicle|
    vehicle.vehicle_reservations.count
  end
end
```

Notice that it's used a LEFT OUTER JOIN in order to accommodate the query conditions for the vehicle reservations.

About about traditional inner joins?

Active Record supports the `.joins()` method and when a traditional inner join is possible, this will likely be the most efficient type of join.

You've now seen a couple of ways to avoid N+1 queries by using Eager Loading.

What if you could configure your application to make it impossible to write N+1 queries altogether?

Prefer Strict Loading Over Lazy Loading

Active Record introduced a new feature called *Strict Loading* in Rails 6.1⁸ that can be used to prevent lazy loading.

Strict Loading can be specified on a per query basis, or globally for an application.

8. <https://github.com/rails/rails/pull/37400>

To better understand how Strict Loading works, take a look at an example using the Rideshare application, continuing to work with the Vehicles and Vehicle Reservations.

Run the following code in bin/rails console.

```
ruby/vehicle_reservations_strict_loading.rb
vehicles = Vehicle.strict_loading.all

vehicles.each do |vehicle|
  vehicle.vehicle_reservations.first.starts_at
end

# `Vehicle` is marked for strict_loading. The VehicleReservation association
# named `:vehicle_reservations` cannot be lazily loaded.
# (ActiveRecord::StrictLoadingViolationError)
```

This time when you run the code you'll get this error:

Vehicle is marked for strict_loading. The VehicleReservation association named :vehicle_reservations cannot be lazily loaded. (ActiveRecord::StrictLoadingViolationError)

When Strict Loading is enabled, Eager Loading must be performed in order to run this query. Try modifying the query above using the Eager Loading techniques you learned earlier to avoid the Strict Loading exception being raised.

You've now worked through one way of removing the possibility of N+1 queries from your Rails application. What other kinds of query-level optimizations can you make?

Optimizing Individual Active Record Queries

Database connections are a finite resource, and all queries require a database connection to run. Reducing unnecessary queries avoids excessive use of limited database connections which can help the scalability of your database server.

What kinds of changes can you make on an individual query basis?

Limiting the fields needed can help speed up queries. One way to do that in Active Record is to use `.select()`, which does not generate a SQL query and uses loaded data. (See: *Rails Pluck vs Select and Map/Collect*⁹)

Another method is `.pluck()`, which generates SQL that specifies limited fields.

9. <https://www.rubyinrails.com/2014/06/05/rails-pluck-vs-select-map-collect/>

Try running this code from bin/rails console:

```
Vehicle.includes(:vehicle_reservations).
  pluck('vehicles.name',
        'vehicle_reservations.starts_at',
        'vehicle_reservations.canceled')
```

`pluck()` is even smart enough with invalid SQL, which requests fields that don't exist on a table, to figure out which tables to get the fields from.

The above query code can be rewritten as below and produce an equivalent result. This may be confusing to maintainers though, and a more the more explicit version above that selects the fields from the appropriate tables is clearer.

```
Vehicle.includes(:vehicle_reservations).
  pluck(:name, :starts_at, :canceled)
```

A major benefit of limiting the fields at query time is unlocking the possibility of Index Scans, and even Index Only Scans, to make retrieval very fast. You'll learn about those soon.

In the next section, you'll see how to save a round trip query by asking for fields to be returned before inserting data.

Save A SELECT With A RETURNING

PostgreSQL has a capability called the RETURNING keyword. It can be used to ask for fields to return before inserting data.

For fields that are populated by the database like a primary key field populated from a Sequence, you might write code to insert data and then query for the id value. Those second queries needed to look up the database supplied value can be replaced using the RETURNING keyword.

Try using the `temp.users` table you set up earlier. The table does not currently have a sequence associated with the `id` column so you'll set that up by running these statements from `psql`:

```
CREATE SEQUENCE temp_users_id_seq INCREMENT 1 START 1;
ALTER TABLE temp.users ALTER COLUMN id
SET DEFAULT nextval('temp_users_id_seq');
```

The first statement adds a sequence and the second statement assigns it to the `id` column. Now `id` will be populated by the sequence when you insert rows.

Try inserting a row, and add the RETURNING keyword asking for the `id` column.

```
INSERT into temp.users (name) VALUES ('bob') RETURNING id, name;
```

You should see the id values in the response without needing to query for them.

Fortunately Active Record supports RETURNING and you can experiment with this using bin/rails console. Insert a single Driver using `.insert_all()` and ask for the id column value:

```
ruby/insert_all_with_returning.rb
Driver.insert_all([
  {
    first_name: 'Andrew',
    last_name: 'Atkinson',
    email: "andrew.atkinson@example.com",
    password_digest: SecureRandom.hex
  }],
  returning: [:id]
)

_._first["id"]
```

You've now used RETURNING to eliminate a query.

How else might you speed up queries?

Bounding Query Results Using LIMIT

Another common way to resolve a poorly performing query is to add boundary conditions. Bounds can be added by filtering to fewer rows or by adding a LIMIT to the amount of rows that are fetched at one time using the Active Record `.limit()` method.

Most queries should be bounded when they are first written, but for small row counts unbounded queries will typically perform OK, and then slow down or not run at a certain row count size. The tipping point will depend on the queries and the server resources.

Another way to add boundaries is to use the Active Record batch operations like `.find_each()`, `.find_in_batches()`, and `.in_batches()` (See: *Rails — `find_each` v.s `find_in_batches` v.s `in_batches`*¹⁰).

These methods are typically used to reduce the amount of memory consumed at once, since each result row turns into an Active Record instance. However these methods also add limits to queries that can help them run reliably even as row counts increase.

10. <https://medium.com/lynns-dev-blog/rails-find-each-v-s-find-in-batches-v-s-in-batches-d5ca9bfe37d>

You've now seen some basics for individual query optimization. As your queries become more complicated, you want to reach for more advanced techniques.

Advanced Query Support In Active Record

*Subquery Expressions*¹¹ or *Subqueries* are what PostgreSQL calls multiple SQL queries combined into one statement. The general pattern is that an outer query uses a second inner query as a way to filter rows down. Filtering is done using a WHERE clause or IN clause. Subqueries are common in SQL and recently support was added to Active Record.

You'll adapt an example from the PostgreSQL Tutorial on Subqueries¹² to Rideshare creating an analytical type of query to show what's possible in a single statement. You'll first write the query in SQL and then convert it to Active Record. The query has these goals:

- Find the average number of Trips for all Drivers
- Find Drivers with more Trips than average
- Order above average Drivers by Trip count in descending order

To find the average in SQL, use the AVG() aggregate function. If you'd been working in bin/rails console, go back to psql for these since you're running SQL queries.

```
sql/select_avg_trips_count.sql
SELECT AVG(trips_count) FROM users
WHERE type = 'Driver';
```

Use the average Trip Count as *input* to a WHERE clause outer query, by making the Trip Count query an inner query or subquery. Modify the query below to find drivers above and below the average.

```
sql/trips_count_subquery.sql
SELECT
  id, trips_count
FROM users
WHERE type = 'Driver'
AND trips_count > (
  -- the earlier AVG(trips) query
  SELECT AVG(trips_count) FROM users
  WHERE type = 'Driver'
)
ORDER BY trips_count
DESC LIMIT 5;
```

11. <https://www.postgresql.org/docs/current/functions-subquery.html>

12. <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-subquery/>

You now have a list of your above average drivers by trip count, in descending order based on the trips they've provided.

How can you adapt this Subquery to Active Record?

Taking inspiration from *Advanced Active Record: Using Subqueries in Rails*¹³ on the PgAnalyze blog, take the techniques and write or review the Active Record code below. Run this code in bin/rails console and confirm that the generated SQL is the same.

```
ruby/active_record_subquery.rb
Driver.where('trips_count > (:avg)', avg: Driver.select('avg(trips_count)').
  order(trips_count: :desc).
  limit(5))
```

In this code, :avg is a placeholder and the value is supplied in a Hash key and value style. Instead of SQL query result rows, you're now working with Active Record objects. Pretty cool.

Subquery Expressions in SQL are very useful but they can become difficult to read. Another advanced query design technique called Common Table Expressions (CTEs) helps make multi-query statements better organized and easier to read and maintain.

Using Common Table Expressions (CTE)

Common Table Expressions abbreviated as a CTE,¹⁴ are a technique in PostgreSQL to combine two or more queries into a single statement with named parts.

CTEs are also called WITH queries because they use the WITH keyword. They'll be referred to as CTEs in this chapter.

CTEs combine queries that are independent but can be used to work together to pull data from multiple places. CTEs can be used recursively by combining a non-recursive portion and a recursive portion and then using a UNION to combine them. Recursive CTEs are beyond the scope of this chapter, but the PostgreSQL Tutorial has a nice example showing how to find direct reports for a manager using the technique.¹⁵

In this section, you'll create a CTE to answer questions about Rideshare Drivers. You'll look at new Drivers on the platform. As you did in the last

13. <https://pganalyze.com/blog/active-record-subqueries-rails>

14. <https://www.postgresql.org/docs/current/queries-with.html>

15. <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-recursive-query/>

section, you'll write the CTE first with SQL and then look at how to translate a CTE into Active Record.

First, write a SQL query without a CTE that finds Drivers created in the last 30 days. From psql connected to the Rideshare database, run this SQL query:

```
sql/trips_new_drivers.sql
-- Drivers created in the last 30 days
SELECT *
FROM users
WHERE created_at >= (NOW() - INTERVAL '30 days');
```

Imagine a Rideshare feature that shows ratings for new Drivers to driver operations managers. To fetch the data, use the rating field on the trips table.

Your CTE will have named query parts so that the query stays organized and maintainable. To populate a lot of rated trips, run `bin/rails data_generators:trips_and_requests` from your terminal. Once you've got a lot of rated trips available, enter this SQL statement into psql:

```
sql/trips_top_drivers_cte.sql
WITH new_drivers AS (
  SELECT *
  FROM users
  WHERE type = 'Driver'
  AND created_at >= (NOW() - INTERVAL '30 days')
),
rated_trips AS (
  SELECT id, driver_id
  FROM trips
  WHERE rating IS NOT NULL
)
-- Display Driver names and
-- the Average rating for their trips
SELECT
  trips.driver_id,
  CONCAT(users.first_name, ' ', users.last_name) AS driver_name,
  ROUND(AVG(trips.rating), 2) AS avg_rating
FROM trips
JOIN users ON trips.driver_id = users.id
JOIN new_drivers ON trips.driver_id = new_drivers.id
JOIN rated_trips ON trips.id = rated_trips.id
WHERE users.type = 'Driver'
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 10;
```

Let's break down what's happening in the CTE above. The CTE has named parts `new_drivers` and `rated_trips` which are each independent queries being brought together to combine Driver data and Trips data. With the data sources

available, the Drivers are then presented with their names and ratings, in descending order showing the top rated drivers first.

Support for PostgreSQL CTEs has been added to recent versions of Active Record in Ruby on Rails. Rails 7.1 adds a `.with()` method (See: *Rails 7.1 Construct Common Table Expressions*¹⁶) to generate a CTE statement. Prior to Rails 7.1 creating a CTE could be done using the `activerecord-cte`¹⁷ gem.

To get some hands on experience with that, you'll adapt a simpler CTE to Active Record. From your terminal, run `bin/rails console` and use this Active Record code:

```
ruby/active_record_cte_with.rb
# use the "." endless range
Trip.with(recently_rated:
  Trip.where.not(rating: nil).
  where(created_at: 30.days.ago..)
).
from("recently_rated").
count
```

Notice how this example combines multiple Active Record style queries into one. The `.with()` method is used as an outer query portion. The name `recently_rated` is given as a name to an inner query portion. Review the generated SQL. This query only counts rows, but consider using the techniques to create the earlier Driver SQL CTE as Active Record.

You've now seen how to perform Subqueries and CTEs in both SQL and Active Record.

As queries become more complex, it can be nice to encapsulate the details of complex queries and provide a simpler interface. Fortunately such a capability exists in PostgreSQL and it's called Database Views. Read on to see how those work and how to manage them with Ruby on Rails.

Introducing Database Views for Rideshare

In this section you'll work with *Database Views* which are a built-in object type for PostgreSQL.

Briefly, database views are a way to name and encapsulate a query and store that as an object within PostgreSQL.

16. <https://blog.kiprosh.com/rails-7-1-construct-cte-using-with-query-method>

17. <https://github.com/vlado/activerecord-cte>

Database Views are not currently supported natively by Active Record. However, in this section you'll create views to use with Rideshare. Since views have a definition that you may want to evolve over time, you'll manage the lifecycle of views using Rideshare. PostgreSQL does not offer a built-in way to manage versioned views over their lifecycle, so using the client application to do this provides a solution.

For Rideshare you'll use a third party Ruby gem that extends Active Record, adding support for Database Views.

Some of the new terminology in this section is listed below.

Administration Terminology



- Database View — Encapsulates a SQL query
 - Materialized View — A View with pre-calculated results to query
-

A database view encapsulates (Wikipedia: *Encapsulation*¹⁸) a SQL query and gives it a name. Imagine that you frequently queried New Drivers in Rideshare and wanted to have a shared definition of what a new driver is as a query that could then be re-used for other purposes. You might create a view called “new_drivers” with a query like this one:

```
CREATE VIEW new_drivers AS
  SELECT * FROM users
  WHERE type = 'Driver'
  AND created_at >= (NOW() - INTERVAL '30 days');
```

With that view created in PostgreSQL, you're now able to write a very simple query, `SELECT * FROM new_drivers;`

Unless views are Materialized, they execute live queries. What does Materialization mean? A Materialized view is a type of View that means the query can be run in advance and then queries to the view – like the one above that queries the pre-calculated results instead of running a live query.

Since the query results are pre-calculated with materialization, querying a materialized view is very fast. The trade-off is that the query results become stale. Fortunately Materialized Views can be refreshed `CONCURRENTLY`. Let's see how to create both types of views and keep materialized views refreshed.

Database views have a definition and you may want to evolve the definition over time. While there's no built-in solution to PostgreSQL or Ruby on Rails

18. [https://en.wikipedia.org/wiki/Encapsulation_\(computer_programming\)](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

for this, you'll use a third party gem to create and manage versioned views. Read on to learn more.

Creating the Search Result Model With Scenic

The third party Ruby gem scenic¹⁹ can be used to create and manage versioned views.

This gem works in the same way as Active Record where the developer writes Ruby code like `create_table` which then produces SQL DDL like `CREATE TABLE`.

The Scenic gem includes a generator that generates two files. The first file is a versioned plain text file with a `.sql` extension where you'll place the SQL definition for your view. The second file is a Migration that adds the view to the database.

For Rideshare, Scenic was used to create a database view to help when searching for Drivers. Open `psql` and run the `\dv` meta command to list the existing views.

Rideshare has a `search_results` database view that's connected to a `SearchResult` Ruby object in the codebase:

```
ruby/search_result.rb
class SearchResult < ApplicationRecord
  belongs_to :searchable, polymorphic: true

  # this isn't strictly necessary, but it will prevent
  # rails from calling save, which would fail anyway.
  def readonly?
    true
  end
end
```

The `SearchResult` class acts as an interface to the database view. To try it out, load `bin/rails console` and run `SearchResult.first`. This will execute the view with a live query and map results into Ruby object instances. The SQL query will be logged in the `bin/rails console`.

The `SearchResult` model interaction for querying it works the same way as other Active Record classes that are backed by database tables and not views.

Using scenic, the definition of the view is versioned and can be changed over time.

19. <https://github.com/scenic-views/scenic>

Views can also be materialized and refreshed concurrently. Read on to learn more.

Improving Performance With Materialized Views

Views do not have a performance advantage when they're not materialized. They do help encapsulate complex SQL queries and can be used to create Active Record object instances backed by a view instead of a database table using Scenic. These benefits are great, but there's also another very big benefit from a performance perspective you'll work with next.

Materialized Views have great performance advantages. When application code can use Materialized Views, very high performance query code can be written because the views have pre-computed query results. Query result execution can be done concurrently as well, which you'll see how to do. This means clients can keep querying the pre-computed results while the view is recalculated in the background and replaced.

To improve performance even further, as is the case with database tables, Materialized views can have Indexes added to them. By writing queries that use Indexes on your Materialized Views you can build very high performance query code.

In the last section you worked with the `SearchResult` model and `search_results` view but this view was not materialized.

In this section you'll work with a new model called `FastSearchResult`.

This model was generated using the `--materialized` flag provided by the `scenic` gem that creates a Materialized View to back the model. The view was generated by running this command:

```
bin/rails g scenic:model fast_search_results --materialized
```

Both views have been added to the `db/structure.sql`. Search in the file for `CREATE MATERIALIZED VIEW` to view the definition. The `db/structure.sql` file shows only the definition of the view and not the precomputed query results.

Viewing the definitions of Materialized Views from `psql` is a bit different. Check the `pg_matviews`²⁰ System Catalog to inspect the definition:

```
SELECT * FROM pg_matviews;
```

The field `ispopulated (bool)` shows whether the view has been populated, and `hasindexes` shows whether any indexes have been added.

20. <https://www.postgresql.org/docs/current/view-pg-matviews.html>

To get the performance advantages you'll want to regularly refresh your materialized views, which can be done both from the application code or with SQL.

The `FastSearchResult` model has a `refresh` method. From `bin/rails` console, refresh the view like this:

```
FastSearchResult.refresh
(20.7ms) REFRESH MATERIALIZED VIEW "fast_search_results";
=> #<PG::Result:0x0000000115bc40d8 status=PGRES_COMMAND_OK \
    ntuples=0 nfields=0 cmd_tuples=0>
```

Compare the response times between `SearchResult.first` and `FastSearchResult.first`. The materialized view will return much faster!

Rideshare keeps the materialized view results refreshed using a cron job. Ruby on Rails does not offer native cron job scheduling support so the third party gem `whenever`²¹ is used to generate and manage jobs.

Review `config/schedule.rb`. This code uses the Rails runner²² command to invoke the same code you called earlier from `bin/rails` console.

```
every 15.minutes do
  runner "FastSearchResult.refresh"
end
```

Materialized Views can be refreshed concurrently as well, although they do require a Unique index to be added. Rideshare added a Unique index to `fast_search_results` to enable concurrent refreshes.

To refresh concurrently from `psql`, run this statement:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY "fast_search_results";
```

You've now seen how to leverage database views with high performance materialized views, that can be indexed and refreshed concurrently.

Besides these techniques, Ruby on Rails adds caching layers for database queries that help improve your application performance. Let's see what types of caches are available and how to leverage them.

Reducing Queries With Active Record Caches

Active Record has a variety of caches that run to improve the performance of your Rails application code.

21. <https://github.com/javan/whenever>

22. https://guides.rubyonrails.org/command_line.html#bin-rails-runner

The first one is the *Query Cache*²³ which is a cache of the SQL statements. Query Cache entries only live for the duration of a Rails controller action.

Loading a Rider's first Trip Request at the beginning of an action with `TripRequest.where(rider_id: 1).first` will place the query and result into the Query Cache. The second time the same query is run inside the same controller action, the query will match against the cached version and the result will be returned from cache instead issuing a second database query to PostgreSQL.

In this way, Active Record helps you avoid unnecessary queries with the Query Cache. To see this in action, take note of CACHE entries in your Rails server log and execution times printed as (0.0ms). The CACHE statements in the Rails log can also be disabled (See: *Disable Rails CACHE Logging*²⁴).

A trade-off with the Query Cache is that more memory is used hold the query and the result. Memory is used to instantiate Active Record objects and their relationships populated from the Query Cache (See: *Cached queries guidelines*²⁵).

While being a useful feature, an over reliance on the Query Cache usage from excessive memory consumption could be an opportunity to improve your database queries making them faster and lighter.

In the next section you'll see PostgreSQL Prepared Statements²⁶ and how to work with them in Active Record.

Prepared Statements With Active Record

Prepared Statements²⁷ are a type of object in PostgreSQL that take a specific query and separate the parameters that are bound to it.

Prepared Statements give a name to the query without parameters. Prepared statements can be viewed by querying the system view `pg_prepared_statements`. Try running `SELECT * FROM pg_prepared_statements;` from `psql` to see what's stored in there.

Prepared Statements are enabled by default and store up to 1000 statements per connection with Active Record.

23. https://guides.rubyonrails.org/caching_with_rails.html#sql-caching

24. <https://heliom.ca/blog/posts/disable-rails-cache-logging>

25. https://docs.gitlab.com/ee/development/cached_queries.html

26. <https://www.postgresql.org/docs/current/sql-prepare.html>

27. <https://guides.rubyonrails.org/configuring.html>

To use Prepared Statements, do not set `query_log_tags_enabled`.

From `bin/rails` console try running some Active Record queries.

```
User.first
```

Explore the saved prepared statements for your connections by running this code:

```
ActiveRecord::Base.connection.execute(
  'SELECT * FROM pg_prepared_statements'
).values
```

Prepared Statements have some incompatibilities so you may need to disable them. Besides being incompatible at publication time with Query Log Tags, they also don't work with some types of connection pooling and PgBouncer, which you'll work with in an upcoming chapter.

Ruby on Rails keeps innovating with prepared statements though. Prior to Rails 7, `SELECT*` was cached when fields weren't requested. In Rails 7 a setting `enumerate_columns_in_select_statements` was added²⁸ that enumerates the list of columns for `SELECT` statements, helping increase re-use of the prepared statement cache.

You've now seen how to use the Query Cache and Prepared Statements. Let's continue the cache theme, and look at Counter Caches.

Eliminating Slow Count Queries With Counter Caches

Another type of Active Record cache is aimed at improving slow `COUNT()` queries by using a pre-computed result instead.

This cache is called the *Counter Cache*. Counter Caches in Active Record keep a running tally of counts and store the tally in an integer column on a table.

In Rideshare, an example `COUNT()` query might be the number of trips a Driver has performed. This query could be done in high volume if many Riders are trying to access the trips count. Take a look at the following Active Record code to get the count of trips for a Driver.

```
Driver.first.trips.count
```

This code can be improved with a counter cache. First, this code is using `.count()` which always issues a SQL `COUNT()` query. Instead of `.count()` prefer the `.size()` method because it counts objects in memory when they're already loaded instead of issuing a SQL `COUNT()` query.

28. <https://www.bigbinary.com/blog/rails-7-adds-setting-for-enumerating-columns-in-select-statements>

Besides counting objects in memory, `.size()` also uses a Counter Cache column when it's available which avoids issuing a `SQL COUNT()` query.

Counter Cache columns include various trade-offs. They require a new column be added and they also add a small bit of latency at write time. Counter caches do open up the possibility of inconsistent counts. A high volume of `UPDATE` statements to the same counter cache column can also cause database bloat which you'll learn more about later.

Despite these downsides, counter caches are a great performance tactic in many situations because they provide nearly instant counts without `SQL COUNT()` queries. Read more in *Caching counters with ActiveRecord's counter caches*.²⁹

The following Migration for Rideshare added a `trips_count` column to the `users` table to act as a counter cache.

```
sh/add_counter_cache.sh
# create the migration
bin/rails g migration AddTripsCountToUsers trips_count:integer

# run the migration
bin/rails db:migrate
```

The `counter_cache: true` option was added to the `:driver` association in the `Trip` model:

```
class Trip < ApplicationRecord
  belongs_to :driver, counter_cache: true
end
```

When adding a new counter cache to a production system, the counter cache will not be populated for existing rows. You'll need to populate the cached value before you can use it or it will return 0.

To explore how the Counter Cache works for a driver, run the following code from `bin/rails` console.

```
ruby/reset_counters.rb
driver = Trip.first.driver
driver.trips.count
Driver.reset_counters(driver.id, :trips)
driver.trips.size
```

This code gets a `Driver`, and queries for the `Trips` count. Notice it makes a `SQL` query. Next it resets the `Trips Count Counter Cache` for the `Driver`. By

29. <https://blog.appsignal.com/2018/06/19/activerecords-counter-cache.html>

then replacing `.count()` with `.size()` and calling `trips.size` repeatedly, notice how no SQL queries are performed to fetch the value. `driver.trips.size` is equivalent to reading the field `drivers.trips_count` directly, which was populated earlier.

You’ve now seen a variety of ways to use Caches to improve your query performance.

Another type of poor performance can be conducting calculations in Ruby code. PostgreSQL has a rich variety of calculation methods available and may be a better place to conduct this work. These methods are available to use with Active Record.

Performing Aggregations In the Database

One opportunity to improve Active Record performance is to perform calculations in PostgreSQL instead of in Ruby and Active Record code, by using built-in SQL aggregate functions.

Database aggregate operations will generally provide better calculations faster due to fewer round trips between the application and the database.

PostgreSQL aggregate functions and Active Record equivalents are displayed below.

	Active Record	SQL
Count	<code>.count()</code>	<code>COUNT()</code>
Average	<code>.average()</code>	<code>AVG()</code>
Minimum	<code>.minimum()</code>	<code>MIN()</code>
Maximum	<code>.maximum()</code>	<code>MAX()</code>
Sum	<code>.sum()</code>	<code>SUM()</code>

Table 1—Aggregations Comparison

Equivalent Aggregations in Active Record and SQL

Active Record support is provided in the `ActiveRecord::Calculations`³⁰ module. For PostgreSQL, refer to documentation on *Aggregate Functions*.³¹

PostgreSQL has more Aggregate Functions beyond those supported by Active Record. PostgreSQL even allows users to make their own aggregate functions called *User-Defined Aggregates*.³²

30. <https://api.rubyonrails.org/classes/ActiveRecord/Calculations.html>

31. <https://www.postgresql.org/docs/current/functions-aggregate.html>

32. <https://www.postgresql.org/docs/current/xaggr.html>

You've seen how to improve your Active Record query code in a variety of ways. Now you'll explore how to directly write SQL within Active Record.

Reduced Object Allocations With SQL In Active Record

While writing Active Record query code typically involves writing Ruby, Active Record also supports SQL. This is a hybrid strategy and is a useful technique where memory or speed is very important.

Active Record has a number of methods that support direct SQL. The first one you'll look at is `.find_by_sql()`. With this method, instead of Ruby Active Record code, you may use SQL as text as an argument.

The results are mapped to instances of the calling class and stored in a Ruby Array. Normally Active Record returns results in a `ActiveRecord::Relation` which enables lazy evaluation and chaining. One trade-off with `.find_by_sql()` is that you won't get lazy evaluation or be able to chain more methods on.

Try running this code from bin/rails console:

```
TripRequest.find_by_sql("SELECT * FROM trips LIMIT 1")
```

Another way to run SQL in Active Record is with the `.execute()` method. With this method the results are mapped to “primitive types” in Ruby like the String or Integer type.

Experiment with this method by running this code from bin/rails console:

```
ActiveRecord::Base.connection.execute("SELECT * FROM trips LIMIT 1")
```

The results are placed into a `PG::Result` class instance. This is a bit lower level than Active Record. Take note of fields like `ntuples=1` and `nfields=7` which indicate one row (tuple) was returned with 7 fields of data.

When primitive types are acceptable or even desired for the response, `.select_all()` and `.select_one()` can be good options.

`.select_one()` works like `.select_all()` but returns only the first item.

With primitive types you won't incur the performance penalty of instantiating Active Record objects because that process does not happen. You'll need to determine whether the results from these methods are compatible with where you wish to use the result data.

To better understand the memory savings opportunity, use Ruby benchmarking techniques as demonstrated in *Optimizing Active Record queries*³³ to

33. <https://dev.to/kattyacuevas/optimizing-active-record-queries-4i84>

compare different ways of fetching and using row data with Active Record. The information below is a performance sample collected using Benchmark.memory, comparing Active Record and SQL equivalents instantiating Trip objects.

Comparison:

```
Using select_one:      4660 allocated
Normal Trip.first:    1988419 allocated - 426.70x more
```

Active Record allocates 426 times more objects compared with the SQL approach. The SQL approaches do not create Trip model instances so the latency involved with that could be avoided.

Wrapping Up

You now have several ways to optimize your Active Record code, starting from improved observability connecting your SQL queries to the Active Record query code that generated it.

You removed unnecessary queries from Active Record helping, conserve limited database resources.

You learned how to identify the N+1 query pattern manually, automatically, and how to fix it using Eager Loading or avoid it using Strict Loading.

You created Subqueries, Common Table Expression (CTE) queries, and Database views. You learned how Views can be Materialized to provide performance advantages. You saw how the lifecycle of versioned database views can be managed over time.

You learned about caches provided by Rails, including the Query Cache and Counter Cache.

With Active Record covered thoroughly, it's time to shift your focus away from the Rails application that generates SQL queries, and into query level observability once those queries have arrived to PostgreSQL.

In the next chapter you'll explore SQL query level performance observability and improvements. Read on to get started.

Improving Query Performance

In this chapter you'll analyze SQL queries running on PostgreSQL, developing a richer understanding of their characteristics as they relate to performance.

To do that you'll learn how to generate and read query execution plans, and use statistics that the database continually collects. Once you understand which portions of a query are most costly, you can begin to use tactics that reduce the costly portions which can improve the overall scalability of your server instance.

Query optimization is a complex subject, with entire books dedicated to covering it. In this chapter you'll get equipped with the basics, learning where to find information and how to interpret it.

Review the following terminology to learn about it for the first time or get refreshed on it. You'll be working with terms like *selectivity*, *cardinality*, a variety of Index types, filtering, sorting, and more in this chapter.

Query Performance Terminology



- Selectivity — How selective a query is
- Cardinality — How many unique values there are
- Sequential Scan — Reading all rows in table
- Index Scan — Fetching values from an Index on a table

To get insights from your query execution, you'll also work with your logs. Before going to PostgreSQL logs, what kind of query information can you get from Active Record logs?

Logging Slow Queries With Active Support Notifications

With Ruby on Rails and without any extra gems, plugins, or database extensions, a wealth of query information can be added to the Rails application log. *Active Support Notifications*¹ can be used to monitor slow queries. (See: *Track slow DB queries without additional gems*²)

To see this in action, read and understand the *Slow Query Subscriber* class added to Rideshare that uses Active Support.

The *Subscriber* listens for events formatted as `sql.active_record` that have query information. The duration of the query can be calculated using the start and finish times associated with the event. When the duration exceeds 1 second, the full SQL statement can be logged using the Rails logger.

Review the class in Rideshare or copy this:

```
ruby/slow_query_subscriber.rb
# Inspiration: https://twitter.com/kukicola/status/1578842934849724416
class SlowQuerySubscriber < ActiveSupport::Subscriber
  SECONDS_THRESHOLD = 1.0

  ActiveSupport::Notifications.subscribe('sql.active_record')
  do |name, start, finish, _, data|
    duration = finish - start

    if duration > SECONDS_THRESHOLD
      sql = data[:sql]
      Rails.logger.debug "[#{name}] #{duration} #{sql}"
    end
  end
end
```

Open `bin/rails` console to test this out. By running the query `SELECT pg_sleep(1);` from Active Record as a query that simulates taking 1 second or more, you'll see the instrumentation get triggered and the following log output.

```
irb(main):004:0> ActiveRecord::Base.connection.execute("SELECT pg_sleep(1)")
[sql.active_record] 1.005035 SELECT pg_sleep(1)
(1005.2ms) SELECT pg_sleep(1)
```

In this technique you've logged a slow query from the client application side. What about when you want to view slow queries from the server side?

1. https://guides.rubyonrails.org/active_support_instrumentation.html
2. <https://twitter.com/kukicola/status/1578842934849724416/photo/1>

Capture Query Statistics In Your Database

The queries in your database consume resources and it's helpful to focus your optimization efforts on the most costly queries. To do that you'll want to get a global view of all queries to see which ones are most costly. You'll use the `pg_stat_statements`³ module which you configured in an earlier chapter, and will be referred to as PGSS.

PGSS takes each query and removes the parameters from it, creating a sort of query group or pattern. Each query then can be put into this group or pattern, and statistics can be collected at the group or pattern level. PGSS then makes the statistics available with a system catalog view.

These groups or patterns are called the *normalized* form of the query. Each normalized query gets a Query Identifier (queryid). The queryid uniquely identifies the original query text before the normalization process. The parameters are removed in the normalization process and replaced with placeholder characters.

Part of the purpose of the normalization where parameters are removed, is so that queries with the same structure can be grouped together. Statistics can then be computed across queries that are similar, sometimes referred to as a “group”.

Query statistics are at the group level including the duration and the number of calls, across all instances of a query within the group that may have very different sets of specific parameters.

Enabling the PGSS module requires a database restart. PGSS is generally recommended to add for all databases. Although it adds a minimal amount of latency to collect statistics, the information it provides is worth it. PGSS is supported by cloud providers of PostgreSQL. If you don't already have it enabled in your production database, plan a time to make this parameter change and restart your database during a low activity period.

To configure the module, add `pg_stat_statements` to `shared_preload_libraries` in `postgresql.conf`. Once that's done and PostgreSQL has restarted, you'll enable the extension.

To enable the extension, run the following statement from `psql`. This only needs to be done once per database.

3. <https://www.postgresql.org/docs/current/pgstatstatements.html>

```
sql/create_extension.sql
```

```
CREATE EXTENSION pg_stat_statements;
```

You may also use `ALTER SYSTEM` to modify `shared_preload_libraries` from a `psql` prompt but this isn't recommended. This method generates a value in `postgresql.auto.conf` that overrides the value in `postgresql.conf` (Thanks to Lukas Fittl for this tip!⁴).

This can lead to confusion about which value is active, so consider making all changes only in your PostgreSQL config file and keeping a version controller backup of your single config file.

Rideshare enabled PGSS by enabling the extension via a Rails Migration.

Now that you've enabled PGSS, read on to find out how to populate and review query statistics for Rideshare.

Rideshare Query Statistics

Since Rideshare isn't a running system, you'll need to simulate application activity by populating some query information.

To do that, start up a Rideshare server by running `bin/rails server` from your Terminal.

In another terminal window, run `bin/rails simulate:app_activity` to simulate activity. You'll should see queries being logged to your server. Since PGSS was enabled earlier, when those queries arrived in PostgreSQL, the denormalization and statistics collection process was happening in the background.

PGSS tracks 5000 normalized queries by default. To track more than that, raise the `pg_stat_statements.max` value. The least-executed queries are discarded when the max is reached.

From `psql`, query `SELECT * FROM pg_stat_statements;` to view everything PGSS has collected. A more useful query, though, might be to look at the top few slowest queries.

Once you've done that, PGSS should have captured some statistics.

From your terminal, run `psql --dbname rideshare_development` again and then use the following query to look at some of the queries and fields of data from PGSS. You may want to run `SELECT pg_stat_statements_reset();` to reset the statistics and run the Rideshare simulation again to isolate the results to being only from the simulation.

4. <https://www.postgresql.org/docs/current/config-setting.html>

```
sql/ten_worst_queries.sql
```

```
SELECT
  total_exec_time,
  mean_exec_time,
  calls,
  query
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;
```

You should now see some Rideshare queries in the PGSS results. A result row will contain `total_exec_time`, `avg_ms`, `calls`, and the normalized query like in this example:

```
sql/pg_stat_statements_result.sql
```

```
-- -[ RECORD 1 ]-----
-- total_exec_time | 2903.9372489999996
-- avg_ms          | 580.7874497999999
-- calls           | 5
-- query           | SELECT "users".* FROM "users" WHERE "users"."type" = $1
```

This is great. You can now review statistics collected database-wide for Rideshare queries. You're accessing these results from `psql`.

When you work on a team of application developers who might not want to use `psql` to view this information, how can you more easily expose it to them?

Introducing PgHero As a Performance Dashboard

The PgHero tool allows you to review query performance information with your team. PgHero is a PostgreSQL Performance Dashboard. PgHero is an open source Rails Engine that ships as a Ruby gem available on GitHub.

When PGSS is enabled and configured, PgHero captures this information and stores it in tables that are presented in a web application view. This can make the information much more accessible to team members that aren't viewing raw PGSS information from `psql`.

PgHero presents the following data for each query:

- Total number of calls
- Average duration
- The percentage the query represents among the total number of query executions

The PgHero percentage makes it easy to scan for queries that are consuming a lot of resources.

The `pghero` gem was added Rideshare. Run `bin/rails server` from your terminal and visit `/pghero` in your browser to explore it.

Once loaded in your browser, navigate to the *Queries* tab. The queries tab shows information collected from PGSS. The following table is a representation of the data that's presented.

Total Time	Average Time	Calls	
1 min 41%	752 ms	58	postgres
SELECT "users".* FROM "users" WHERE "users"."type" = \$1			

Table 2—PgHero Query Statistics

Normalized query statistics from `pg_stat_statements`

PgHero offers a built-in EXPLAIN (ANALYZE) interface. Queries are safe to execute here because they're placed inside a transaction that's rolled back. A `SET LOCAL` statement timeout of 1 second adds another safeguard.

```
sql/set_local_statement_timeout.sql
SET LOCAL statement_timeout = 1000;
```

The PgHero *Space* section shows the largest tables and indexes. Unused indexes have an *Unused* label on them. This screen is easy to scan and visually identify unused indexes that could be removed. Index maintenance will be covered in an upcoming chapter.

The PgHero *Live Queries* screen shows currently executing queries. This screen uses the `pg_stat_activity` system catalog view you saw in an earlier chapter. Queries may also be terminated or canceled from this screen, which is something you've learned how to do using `psql`.

The *Maintenance* tab shows when tables were last Vacuumed and last Analyzed. We'll do more with the `VACUUM` and `ANALYZE` commands later on.

PgHero is open source; it may be forked and extended with more features (See the [andyatkinson/pghero](https://github.com/andyatkinson/pghero)⁵ fork).

Rideshare uses a forked version of PgHero that makes some small additions. One addition adds a column to the estimated bloat percentage screen. Index bloat is covered in more depth later.

The second addition is a feature that adds visibility into background jobs scheduled using the `pg_cron`⁶ extension. You'll learn more about scheduling jobs directly in PostgreSQL later, too.

5. <https://github.com/andyatkinson/pghero>

6. https://github.com/citusdata/pg_cron

You’ve now seen some ways PgHero can be used as a comprehensive PostgreSQL Performance dashboard for your team. PgHero is so useful that you’ll continue to work with it in upcoming chapters. For now, you’ll focus in on Query Performance.

What other query performance PostgreSQL level visibility tools are available?

Once you have broad query level observability, you’re prepared to identify individual queries to optimize. To optimize queries in PostgreSQL, you’ll need to see how PostgreSQL plans to execute the query. To do that you’ll want the EXPLAIN tool.

Analyzing Query Execution Plans

SQL is a declarative language. When you write a SQL statement you declare the results you want and PostgreSQL determines the best way to get them. PostgreSQL shows you a lot of detail about how it plans and estimates what it will do.

The steps PostgreSQL takes are called the Query Execution Plan. Generating and reading Query Execution Plans is critical to understanding queries, identifying issues, and making improvements. This section will introduce or act as a refresher on how to generate and read query execution plans.

Open up a psql prompt and add EXPLAIN in front of any SQL query. Run this example:

```
EXPLAIN SELECT 1;
               QUERY PLAN
-----
Result  (cost=0.00..0.01 rows=1 width=4)
(1 row)
```

EXPLAIN is a tool with many more arguments like ANALYZE, BUFFERS, and FORMAT that you’ll see how to use. The main purpose is to show some details about the plan it has in mind to fulfill your request.

The PostgreSQL query execution planner is also called the *optimizer*. This name is based on how it works. The planner will compare multiple query execution options against each other when that’s possible, and choose the optimal plan.

Here are some basics about planning. The planner creates plans broken into nodes and each node has a cost. The costs are totaled up and the lowest cost plan is selected. This planning happens at runtime when you run your query and does add a bit of latency to the process. The planning time itself is added

to the plan output. Run the example above as `EXPLAIN (ANALYZE) SELECT 1;` to see Planning Time added to the output.

Reading query execution plans is a good skill to develop for multiple reasons. You'll also begin to understand how PostgreSQL parses a query and handles different pieces like selecting fields, filtering rows based on column criteria, or ordering results, and this can inform your query design and schema design.

You may have noticed the Planning Time is extremely short. To help make this possible, PostgreSQL relies on pre-calculated statistics about tables, indexes, and estimates. You may not have realized that PostgreSQL continually collects statistics about all the data in your database. You can even trigger manual statistics collection yourself.

Stats can be collected by running the `ANALYZE` command for a table. From `psql`, try running `ANALYZE trips;` to give this a spin.

It would be infeasible to run `ANALYZE` manually all the time. Fortunately PostgreSQL runs it automatically. `Analyze` runs automatically by being part of the Autovacuum process that is enabled by default in PostgreSQL.

When PostgreSQL is running, and Autovacuum is triggered for a table based on its trigger conditions, a `VACUUM (ANALYZE)` process runs for each table in the database. You haven't yet learned about the trigger thresholds and how to tune them, but you will later. In normal operations and for Rideshare, table statistics are updated automatically and the default values you have in your local development machine are fine for now.

Now you know how Stats are collected and you saw the `ANALYZE` argument with `VACUUM`.

`EXPLAIN` also supports an `ANALYZE` argument but it has a different purpose in that context.

`EXPLAIN` only shows a query execution plan but doesn't run it. To both show the execution plan and run the SQL query, use `EXPLAIN ANALYZE`.

With all of that in mind, it's time to start diving into query execution plans.

Query execution plans are packed with information. Besides the tables, indexes, and columns there are many more types of data. Some of the information is presented below. Some interpretation information and "next step" questions are listed above each type of item for you to begin thinking about. For now, focus more on seeing what kind of information is listed over the interpretation of the information.

- Costs
- Timing
- Scan Types. Sequential Scans, *Index Scans*, Index Only Scans
- Filtering. How many rows were filtered from a larger set?
- Indexes. Which indexes are named and were used? Are there indexes not listed that you expected to be there?
- Loops. For plan nodes, are loops greater than 1? This means the plan node was repeated.
- Ordering. Is sorting visible in the plan? Indexes are ordered; was an Index used for the sort operation?

For any Index Scans, compare the list of fields in the query to the fields that are indexed. Can the fields being selected in the query be reduced? If the selected fields match the indexed fields exactly, a more optimal *Index Only Scan* can be used over an Index Scan.

The BUFFERS argument to EXPLAIN provides insight about the amount of disk data that was fetched. The amount of data fetched can negatively impact query execution speed. Is it possible to select less data, such as fewer rows by using a LIMIT clause?

Look for TIMESTAMP and TIMESTAMPTZ columns in the query and whether those fields are indexed for the table. When these timestamp fields are covered by a B-tree index they are stored in a chronological ordering that can be useful for ORDER BY clauses to reduce the sort cost.

As you begin reading PostgreSQL query execution plans you may want to use additional tools to help you understand them. Free and commercial analysis tools are available. A query plan visualizer is built in to the PgAdmin⁷ open source PostgreSQL GUI client. A free tool is also available at explain.depesz.com.⁸

PgMustard is a commercial tool for query plan analysis. Capture query execution plans from your planner and then upload the plan text to the PgMustard web client to get deep insights from the execution plan. While the tool is commercial, PgMustard does provide a free public EXPLAIN Glossary⁹ that helps those learning all of the terms and concepts the query planner uses.

You've now started to read query execution plans and have seen some of the information you can learn. Next you'll dive into some specific query performance problems.

7. <https://www.pgadmin.org>

8. <https://explain.depesz.com>

9. <https://www.pgmustard.com/docs/explain>

One of the most common query performance problems is inefficiently filtering down rows. For efficient filtering you'll want your database table to have an Index that matches the fields in the filter criteria.

Without those indexes, you've got a *Missing Indexes* class of performance problem. How do you find those and fix them?

Finding Missing Indexes

Missing Indexes are a type of performance problem where a query filters down a large set of rows but does that without Indexes.

Since there's no index, PostgreSQL has to scan the entire table, called a *Sequential Scan*. For tables with a row count in the millions, and when filtering just one or a few rows in your result, scanning the whole table is inefficient.

How can you discover these opportunities to add indexes and make sure they improve the query performance? One technique is to use a query like `find_missing_indexes.sql`¹⁰ to analyze PostgreSQL and what it's tracked about past queries.

Find Missing Indexes shows the counts of Sequential Scans and Index Scans for a given table. When a table has a lot more Sequential Scans compared with Index Scans, the table might be missing indexes.

PgHero even has a *Suggested Indexes* capability you can use to help spot missing indexes. This tool uses row estimates for the table, and analyzes the existing table indexes and queries in order to suggest indexes to add.

Another tool that can be used is a command line tool. This tool is from the `rails_best_practices`¹¹ gem and can be used to help identify missing indexes. The gem has been added to Rideshare. From your terminal, run `bin/rails_best_practices .` (the period means you're running it for the current directory) to see what it suggests. Vehicle Reservations have a Foreign Key column to Trip Requests, and Rails Best Practices suggests adding an index to that column.

Although you may wish to proactively add Indexes to tables, consider balancing that with a periodic removal of unused indexes. This process will be covered more later.

10. https://github.com/andyatkinson/pg_scripts/blob/master/find_missing_indexes.sql

11. https://github.com/flyerhzm/rails_best_practices

Besides *PgHero*, there are commercial tools like PgAnalyze that can help you identify missing indexes. PgAnalyze offers a free tool called *PgAnalyze Index Advisor*.¹²

Besides manually adding indexes, Index creation can even be automated using some advanced tools. The HypoPG¹³ extension and dexter¹⁴ tool can automatically create indexes.

You've now seen some ways to build up your query observability tool set.

While PGSS is a critical tool, one limitation is that it shows you a normalized query. Once you've identified a specific poor query group with PGSS, you will want to gather a sample from production. From the sample of a real execution of a query in that group, you can look at the EXPLAIN plan to see whether there's a missing Index or something else to improve.

Later, you'll keep diving further into query analysis. But right now a piece in the query observability tool chain is missing.

From PGSS, you'll want to collect real query samples. Once equipped with those samples, you'll be able to dive into their details. Let's see a couple of methods to get those samples.

Logging Slow Queries

PGSS is a critical tool, but you'll need sample queries within a group to analyze further.

One way is to log slow queries into postgresql.log. You can do this by setting the `log_min_duration_statement` parameter to a value like 1 second, to log any query that takes longer. From psql run `SHOW data_directory;` to find the data directory path. In this directory you'll find your postgresql.conf file and the log directory where your log file is located. Run `SELECT pg_current_logfile();` to see the current log file.

In the config file, set the following parameter and save the file. Reload your config with `pg_ctl reload --pgdata=DATADIR`, replacing DATADIR with the value from `SHOW data_directory;`.

```
log_min_duration = 1000
```

12. <https://pganalyze.com/index-advisor>

13. <https://github.com/HypoPG/hypopg>

14. <https://github.com/ankane/dexter>

After making that change, reloading your config, and tailing your log file, run `SELECT pg_sleep(1);` from psql to test it out. If your config is working, you'll see the logged SQL statement that took 1 second in the postgres log file.

```
pid=90062 query_id=440101247839410938: LOG:  |
duration: 1000.963 ms  plan:
    Query Text: select pg_sleep(1);
    Result  (cost=0.00..0.01 rows=1 width=4)
pid=90062 query_id=440101247839410938: LOG:  |
duration: 1001.715 ms  statement: select pg_sleep(1);
```

Follow this approach for your production instance and start by logging very slow queries. This will be a way to begin capturing full queries with parameters using your logs.

Once you have a full query with parameters, you'll want to capture a query execution plan manually by running your query with `EXPLAIN`.

What if you could automatically capture query execution plans?

Automatically Gathering Execution Plans

The PostgreSQL `auto_explain` module can be used to automatically capture query execution plans. You wouldn't want to capture every execution plan, but fortunately it can be configured to capture query execution plans for slow queries.

To limit the execution plan capture to slow queries, you'll open up your PostgreSQL config file again and set another parameter. Set the `auto_explain.log_min_duration` parameter to 1000 as you did earlier for the `log_min_duration_statement`. This means for queries that take 1 second or more the query execution plan will be logged into the PostgreSQL log file.

For Rideshare, queries run much faster. Try lowering it temporarily to 0 so that all queries have logged execution plans.

If you're tailing the log file with `tail -f` and the path to your `postgresql.log`, you'll immediately see the a multi-line logged query execution plan. You'll also see a queryid there, which became more useful in newer versions of PostgreSQL (See: *Using Query ID in Postgres 14*¹⁵). The logged query execution plan should be the same as the one you generated manually using `EXPLAIN`.

With the Query Identifier or queryid you can now connect some of the dots between PGSS, logged queries, and execution plans.

1. Identify the queryid from a normalized query in PGSS

15. <https://blog.rustprooflabs.com/2021/10/postgres-14-query-id>

2. Enable query execution plan logging for slow queries using `auto_explain`
3. Find matching queryid values in the `postgresql.log`

Make sure `compute_query_id` is set to on in your PostgreSQL config file.

Once you've found a match, you have the query as executed with real parameters that you can analyze using PostgreSQL EXPLAIN.

You now have a game plan to find high frequency and slow queries with PGSS, and then either manually or automatically collect samples from the log, including manual or automatic collection of the query execution plan. You haven't yet seen how to use the query plan, but that will be coming soon.

You're just about ready to dive into execution plans.

Before doing that, there's one more thing to consider before working on query optimization. PostgreSQL can require some database maintenance for tables and indexes that are related to the slow query. Before spending a lot of energy on query optimization, make sure the tables and indexes are optimized.

Performing Maintenance First

Before diving too far into query execution plans make sure the tables and indexes related to the query are optimized.

Perform some of the generic database maintenance steps when you're facing a specific slow query.

1. For each table in the query run a manual `VACUUM (ANALYZE)` from `psql` for the table. This makes sure Vacuum has run recently and statistics are updated.
2. For any indexes used by the query, determine whether they have a high percentage of estimated bloat. If so, rebuild the index.
3. If you see Autovacuum or Analyze not running regularly for Indexes with very high bloat across the board, it may be time to schedule supplemental Vacuum, Analyze, and Reindex operations. You'll learn more about that later.

Refer to the *Show database bloat*¹⁶ scripts from the PostgreSQL Wiki. These scripts help you identify indexes with a very high estimated percentage of bloat.

Now that you've made sure the tables and indexes are in good shape, you're ready to dive into query optimization.

16. https://wiki.postgresql.org/wiki/Show_database_bloat

Interpreting Index Scan Execution Plan Info

One of the main use cases for Indexes is efficient lookup and filtering. When a column `user_id` is listed in a `WHERE` clause, make sure there's an index for the table covering `user_id` either as a single column index, or as part of a Multicolumn index where `user_id` is the first column listed.

PostgreSQL won't always choose to use your index until row counts are high, and when a single row or a small set of rows are being requested. PostgreSQL uses the selectivity of the query, cardinality for the column, and compares the costs between scanning the whole table, fetching from the index and table, or using the index alone to determine what's best.

When the index is used, the planner shows an *Index Scan* and lists the name of the index.

The basics of index data structure internals are as follows. Each item in the Index is a reference to a table row. Scanning an Index is typically a faster way to collect a list of row pointers compared with scanning all rows for a large table.

Table access is slower because the data is spread out in random locations. Indexes, on the other hand, are ordered and designed for fast retrieval. Because of this ordered storage characteristic, besides filtering, Indexes can also be used for sorting with `ORDER BY` operations.

The values of a node within the index include a `ctid`. The `ctid` is a pointer to the disk location of the row data. You can add the `ctid` to your query to see the value.

Besides Index Scans, there is an even more efficient scan type related to indexes.

With *Index Only Scans*, PostgreSQL fetches all of the data it needs from the index itself directly without needing to access the heap where the rows are.

Index Only Scans are not supported for all index types. B-tree indexes always support Index Only Scans. Typically B-tree indexes are used for equality comparisons such as comparing the exact value of `user_id = 1`. An index covering all values of `user_id` will be traversed until the value is found.

For now, Indexes can be put into two major categories. One category is a general purpose index and the second category would be a “specialized” index. A general purpose index is useful for many queries and operation types while a specialized index might be designed to be optimal for one query.

Indexes trade off faster retrieval speed for more disk space consumption. When performance is critical it may be worth creating specialized indexes.

As a brief introduction to specialized indexes, “Multicolumn Indexes” and “Covering Indexes” are specialized index types that can be used as part of your query optimization tactics.

The query planner may scan the entire table as a Sequential Scan or may use an Index and perform an Index Scan. Is there a middle ground between table scans and Index scans? There is – Bitmap Scans split the difference.

EXPLAIN Scan Nodes and Bitmap Scans

Let’s see how Bitmap Scans are used. The *Database Page Layout*¹⁷ page is extensive and beyond the scope of the chapter, but some basics will be covered here.

PostgreSQL table data (the *heap*) is represented in a *Page*. The bitmap PostgreSQL builds is one it creates on the fly by visiting every Page, and marking whether a Page contains a row and should be scanned or not.

The scan type is something PostgreSQL chooses but is exposed in the query execution plan.

Bitmap Scans can refer to either the Heap (table) or an Index. A Bitmap Index Scan (See: *EXPLAIN - Bitmap Index Scan*¹⁸) is like a middle ground between a slow scan of the entire table and a fast Index Scan. When would a Bitmap Index Scan be used and not an Index Scan? One scenario would be when an Index would be useful but it doesn’t exist covering the fields being filtered on.

On the other side of things with very large result sets, PostgreSQL may choose to not use an Index Scan for an index that exists because it might be faster to scan the entire table before filtering.

A Bitmap Heap Scan builds a bitmap data structure from a heap scan. A Bitmap Index Scans builds on top of that.

The bitmap is one bit per heap page. (See: *Understanding “bitmap heap scan” and “bitmap index scan”*¹⁹)

17. <https://www.postgresql.org/docs/current/storage-page-layout.html>

18. <https://pganalyze.com/docs/explain/scan-nodes/bitmap-index-scan>

19. <https://dba.stackexchange.com/a/119391>

The planner will show the cost of building the bitmap with a Bitmap Heap Scan. A Bitmap Index Scan will show in the execution plan as a Scan Node when a Bitmap is used.

In this section, you were introduced to Bitmap Heap Scans and Bitmap Index Scans. When you see them in query execution plans, you'll now a little more about how PostgreSQL is trying to efficiently filter a set of rows. From there you may explore terminology in greater depth and whether an even more efficient method for filtering is possible given a specific query and specific table structure and row count.

Besides inefficient filtering operations, another type of performance problem is selecting too many rows. How can you find and address instances of that problem?

Adding Query Boundaries With Filtering and LIMIT

In Ruby on Rails applications operating at scale, you'll want to make sure queries in high frequency use cases filter to the smallest set of rows possible.

Try to add a LIMIT clause to most queries. Besides a LIMIT, try and add more boundaries to queries. Extra indexed fields helps the planner cut down the initial set of rows to scan.

In a multi-tenant application a `tenant_id` or `customer_id` column could be added to tables and indexed, so that a WHERE clause can be scoped just to those rows.

If your users are spread out among geographic regions or sales regions, consider grouping them into a regions table and scoping their records using an indexed `region_id` Foreign Key column.

Perhaps you can filter on Active and Inactive users.

Besides filtering in the WHERE clause, remember to add more conditions to JOIN operations.

For web applications, sometimes you'll need to have upper boundaries set and then introduce Pagination or Searching as user experience workarounds. You'll explore pagination techniques like Keyset Pagination later.

In the next section you'll move on to counting items efficiently.

Performing Fast COUNT() Queries

COUNT(*) queries are known to be slow, as explained in *Faster PostgreSQL Counting*.²⁰ They scan the whole table for accurate results. Counts might be for all rows in a table or for a filtered set of rows.

You can achieve fast counts for both unfiltered and filtered scenarios, but you'll need to use some tricks.

The easiest trick is to use the count estimates that PostgreSQL tracks for each table instead of running a COUNT() query. These estimates will be off a bit but are incredibly quick to fetch.

Open up psql from your terminal for the Rideshare database. Run the following SQL statements. Since this is counting all tables named users, drop the temp.users table if you still have it, by running DROP TABLE temp.users CASCADE; to get a single count estimate.

```
sql/estimated_count_reltuples.sql
-- Update stats
ANALYZE users;

-- Use reltuples for estimate
SELECT reltuples::NUMERIC FROM pg_class WHERE relname='users';
```

The first statement updates the statistics for the users table and the second statement fetches the estimated row count.

The fast_count²¹ gem has been added to Rideshare, which enables faster counting for Active Record. The gem adds a Migration that adds a database function for fast counts.

For Rideshare, run bin/rails console and try out the User.fast_count method. Compare it with User.count for speed and accuracy.

```
User.fast_count
```

Although you haven't yet worked in detail with Query execution plans, the execution plan text itself can be used as a data source.

This trick uses the query execution plan to get fast estimated filtered counts. Try running bin/rails console in the Rideshare directory and comparing these two Active Record snippets that count filtered sets of Users. If you ran the data generator you should have 10 million users.

```
User.where("last_name ILIKE 'lname123%'").count
```

20. <https://www.citusdata.com/blog/2016/10/12/count-performance/>

21. https://github.com/fatkodima/fast_count

```
User.where("last_name ILIKE 'lname123%'").estimated_count
```

The `estimated_count()` method is defined in the “fast_counts” gem. The estimated count is a bit off from the real count value but is tremendously faster to get.

Finally, if you need to count distinct values, the PostgreSQL wiki documents some tricks on how to do this quickly (See: *Selecting Distinct Values*²²). Fast distinct counts are being worked on as an addition to the fast_count gem at publication time.

Another advanced technique is to use the postgresql-hll extension. The HyperLogLog (HLL) concept is used to estimate the number of distinct values in a large set. The HyperLogLog is described as a *cardinality estimator*.

After installing and enabling the PostgreSQL extension, the HyperLogLog functionality may be used via the active-hll²³ gem. Try configuring it with Rideshare and installing the gem if you’d like to explore it.

If estimates or these tricks won’t work, you may need to manage your own tally of rows at Insert time. The general technique is similar to how the Active Record Counter Cache works.

When reporting multiple COUNT values a CASE statement may be used. The CASE statement is used to set up different integer values and then sum the values.

An alternative to using SUM with CASE and WHEN is the lesser-known FILTER keyword, which can also be used to display multiple counts at once. The post *The FILTER clause in Postgres 9.4*²⁴ shows how to use FILTER for “scoped aggregate functions.”

Try this out for Rideshare. Run this statement from psql:

```
sql/count_with_filter.sql
```

```
SELECT
  COUNT(*) FILTER (WHERE type = 'Rider') AS rider_count,
  COUNT(*) FILTER (WHERE type = 'Driver') AS driver_count
FROM users;
```

The query above displays the counts of Riders and Drivers. Although it scans the table, it does only one scan.

You’ve now seen some ways to improve the performance for count operations. Next you’ll look at some tools that will help you with your database design.

22. https://wiki.postgresql.org/wiki/Loose_indexscan

23. https://github.com/ankane/active_hll

24. <https://medium.com/little-programming-joys/the-filter-clause-in-postgres-9-4-3dd327d3c852>

Using Code and SQL Analysis Tools

Here you'll see how the rails-pg-extras²⁵ Ruby gem and command-line tool can be used to help with schema design and query design. The gem has been added to Rideshare.

In your terminal, run `bin/rails pg_extras:cache_hit` to look at index hit rate and table hit rate ratios. You want to see high hit ratios for both types.

A low index hit ratio could indicate there are too few indexes. PostgreSQL keeps a cache of indexes in memory and a low hit ratio means there are many cache misses. Refer to the gem documentation for more information.

Next run the `bin/rails pg_extras:diagnose` command to get an overall report for Rideshare. Good items are printed in green and improvement areas are printed in red.

Indexes may be skipped for usage in execution plans when they're suffering from very high estimated bloat. As you saw earlier, the solution in that case would be to rebuild the index.

Another analysis tool you saw earlier is `pgbadger`. `PgBadger` can be also be used for historical query analysis.

Queries are presented in P90, P95, and P99 percentile groupings. Focus on the P90 percentile first, before tackling optimization opportunities for less common P95 and P99 percentiles.

Wrapping Up

This chapter focused on helping you build your query observability tool chain. You automatically logged slow queries from the application. You found high impact slow queries using PGSS on the server. You learned how to manually and automatically collect samples and execution plans. You started to identify common performance problems like Missing Indexes and tactics to fix those problems.

You didn't quite yet dive into queries in depth, but you gained comfort modifying parameters and using the PostgreSQL log to identify information. You configured PGSS and `auto_explain` extensions to help you identify and fix problems.

You started to scratch the surface of query execution plans and how Indexing works.

25. <https://github.com/pawurb/rails-pg-extras>

Next you'll cover Indexes offered by PostgreSQL in greater depth. You'll explore general purpose and specialized index types for different types of queries. You'll keep working with query execution plans expanding on what you started with in this chapter.

Optimized Indexes For Fast Retrieval

In the previous chapter you learned how Indexes can be used for filtering operations. Now you're ready to do more with Indexes, going into greater depth on the types of indexes available in PostgreSQL, which fields to cover, and which types are suited best for certain queries. You'll expand on how indexes can be general purpose or have specialized purposes.

You'll create single-column indexes, multicolumn indexes, partial indexes, indexes on expressions, and Covering indexes. As usual, there is a lot of new terminology to learn with Indexing.

Besides speeding up queries for data retrieval from filter operations, indexes are used to enforce UNIQUE, PRIMARY KEY, and EXCLUSION constraints. Indexes store data in sorted order so they can also be used for ORDER BY operations.

Take a look at some of the Indexing terminology you'll encounter.

Indexes Terminology



- Index Definition — How the Index was created
- Multicolumn Indexes — Index definitions that cover multiple columns
- Partial Indexes — Index definitions with an expression that reduces the covered rows
- Covering Indexes — Index definitions that cover all columns for a query

You'll work with some less common Index types, but first up is a refresher on the default Index type, which is the most common type.

The default and most common Index type used is a B-tree Index.¹ A B-tree index is a secondary data structure represented as a tree optimized for retrieval speed.

B-tree indexes are used for equality comparisons. Each node points to a row. A query using the Index may be searching for rows with matching column values to filter down a result set.

When an Index was scanned and used to identify the row, as you saw earlier the query execution plan will name the index and show that an Index Scan was performed. The index contains a reference to the row identifier. Using the row identifier called the ctid, the table or heap is then scanned to retrieve the row with the columns specified in the query.

With that refresher in place, it's time to work with a less common Index that transforms data before being stored.

Generating Data for Experiments

One of the best ways to learn about the behavior of PostgreSQL is to create experiments by generating test data and performing queries and running commands on large sets of data. To work with Indexes locally on large sized data, open up psql from your Terminal and run these SQL statements:

```
sql/insert_users_generate_series.sql
CREATE SCHEMA IF NOT EXISTS temp;

DROP TABLE IF EXISTS temp.users;

CREATE TABLE IF NOT EXISTS temp.users AS
SELECT
  'fname' || seq AS first_name,
  'lname' || seq AS last_name,
  'user_' || seq || '@' || (
    CASE (RANDOM() * 2)::INT
      WHEN 0 THEN 'gmail'
      WHEN 1 THEN 'hotmail'
      WHEN 2 THEN 'yahoo'
    END
  ) || '.com' AS email,
  CASE (seq % 2)
    WHEN 0 THEN 'Driver'
    ELSE 'Rider'
  END AS type,
  NOW() AS created_at,
  NOW() AS updated_at
FROM GENERATE_SERIES(1, 10000000) seq;
```

1. <https://www.postgresql.org/docs/current/btree-implementation.html>

These statements create a temp.users table on the fly from a SELECT statement inside the temp schema you already have or will create. The temp.users table is similar to the Rideshare users table but with fewer columns. You'll add indexes covering some of these columns.

To start, there will be no indexes on the temp.users table. This makes it faster to insert rows because no Index maintenance will be performed as rows are inserted, updated, or deleted.

Once you have the table created and populated with 10 million rows, you're ready to begin creating experiments.

When filtering rows for tables with low row counts, PostgreSQL may decide to scan the whole table instead of using an Index you've created when it needs to find a few rows for a query. This can be surprising behavior but makes sense as you learn more about Indexing.

You may already know that Index Scans will outperform a Sequential Scan (table scan) when filtering for a row or two. You may not know that when items are found in an index and the PostgreSQL query plan shows an "Index Scan", that PostgreSQL doesn't usually get all of the information needed solely from the Index. PostgreSQL still needs to access the "Heap"² (which is the table row data) after the Index most of the time. That means an Index Scan might be two scans, a Index Scan and a Heap Scan. The combination of those two scans might be slower than scanning the table directly. This is dependent on many factors.

The most optimal scan type is an Index Only Scan. This happens when all of the fields needed either from a SELECT clause or a filtering option are contained in the Index itself. In that case a Heap scan is not necessary, which saves a considerable amount of time.

Enabling Index Only Scans is one of the reasons you saw earlier to limit the fields you're requesting in your Active Record query code.

With those explanations in place, you're now ready to begin experimenting with the temp.users table you've just created.

From psql run this query:

```
SELECT * FROM temp.users
WHERE last_name IN ('lname10000', 'lname100000', 'lname1000000');
```

2. <https://www.postgresql.org/docs/current/glossary.html>

Without an index this query might take a long time. From psql, run the query again with EXPLAIN (ANALYZE, BUFFERS) in front of it. On PostgreSQL 15 this uses a Parallel Sequential Scan for the temp.users table, which is quite fast but takes more than 500ms and reads over 100,000 buffers. In a web application context where this query was running at a high volume, this time might be unacceptably slow and cause a lot of unnecessary disk access.

You'll fix this by adding an index to the temp.users table. Since you're filtering on the last_name field, add an index to that field.

Since you're adding the index to a non-live system, skip the CONCURRENTLY option – it's not needed and would slow down the creation unnecessarily. To see how long it takes to add, toggle on timing by running the \timing command. With timing enabled, run this statement:

```
CREATE INDEX users_last_name_idx ON temp.users (last_name);
```

After adding the index, analyze the table using the ANALYZE temp.users statement from psql.

Run the query again using EXPLAIN (ANALYZE, VERBOSE). The query now runs in less than 5 milliseconds and there is only one Buffer that's read. The index you just added made this query very efficient with a very low cost and low disk usage. You'll see the name of the index "users_last_name_idx" and a Index Scan as the scan type.

As a Rails developer, you use Active Record Migrations to create indexes, keeping all of the databases the application works with in sync.

Create an Active Record Migration for this Index Creation. This way the index will be created on your local development machine, in any lower environments like CI or staging, and in your production environment.

An example Migration is listed below. This Migration uses the if_not_exists: true option which adds the index only if it doesn't exist. This way you're able to add the Index manually if necessary to production, but backfill a Migration to keep everything in sync. The Migration will run on a live system so the CONCURRENTLY option is used in the conventional way for Active Record.

```
ruby/add_index_users_last_name.rb
```

```
class AddIndexUsersLastName < ActiveRecord::Migration[7.0]
  disable_ddl_transaction!

  def change
    add_index :users, :last_name, algorithm: :concurrently,
      if_not_exists: true
  end
end
```

If you're newer to Active Record or haven't worked with it as much, you might be wondering how Ruby generates SQL. Read on to learn more.

Transforming Ruby to SQL

As a Rails developer you may have only created indexes using Active Record Migrations and be less familiar with the SQL statements they generate.

In the last section you saw a CREATE INDEX statement in SQL by creating it with SQL first and then an example of how it's created with Ruby.

You may wish to explore the definitions as creation statements for all of the Indexes when removing and adding back Indexes. PostgreSQL tracks the index creation statements internally in system catalogs. Query the pg_indexes catalog from psql as follows to take a look.

```
SELECT indexdef FROM pg_indexes WHERE tablename = 'users';
```

The statement above shows the index definitions as creation statements for all indexes on the public.users table.

Displayed this way, the creation statements do not contain the CONCURRENTLY keyword even though they may have been created using it.

On your non-live system Rideshare database, practice with more SQL Index modifications. Try dropping an index. Remember that your local development database is a great place to practice. You can always reset the Rideshare schema completely to the original configuration.

From psql, run the DROP INDEX statement below to drop an index named “index_users_on_last_name”.

```
DROP INDEX index_users_on_last_name;
```

Dropping Indexes in a live system is also an operation that should use the CONCURRENTLY keyword:

```
DROP INDEX CONCURRENTLY index_users_on_email;
```

If there are other transactions using the index, CONCURRENTLY avoids introducing long-running locks that would block those transactions.

Now that the indexes are dropped, try manually creating them using the SQL statement and the CONCURRENTLY keyword. Make sure that \timing is enabled.

Why Are My Indexes Not Being Used?

As you saw earlier, even when you've added an Index you want PostgreSQL to use, it may choose to not use it. You saw that this can depend on many factors. Let's look at some of those factors in more depth.

Queries should be as selective as possible, meaning they select as few rows as possible, for your database server to scale better and achieve a higher volume of queries. Most of the Active Record queries you're writing or running are in an online transaction processing (OLTP) Ruby on Rails web application context where short durations are important.

Besides very selective queries, consider the cardinality of the columns you're working with. The number of distinct values in a column is the "cardinality." High cardinality means there are a lot of distinct values.

Indexes are ideal for high cardinality columns and highly selectivity queries. When there is a table index for a column with low cardinality, or in other words, the column has very few distinct values such as two or three total possible values, PostgreSQL may prefer a Sequential Scan over an Index Scan. In that case the Index Scan and Heap Scan may have a higher total cost compared with a Heap Scan only.

In the next section you'll begin to consider the definitions of your indexes. Indexes can cover single columns and multiple columns. Read on to learn more.

Using Single Column And Multiple Column Indexes

*Multicolumn Indexes*³ are indexes in PostgreSQL that include multiple columns in their definition. Terminology like "composite" or "compound" indexes from other relational databases are synonyms for Multicolumn Indexes in PostgreSQL.

Open the Rideshare database with `psql`. Run `VACUUM ANALYZE` without listing any tables to run it for all tables. Use the Index Maintenance query from the PostgreSQL wiki https://wiki.postgresql.org/wiki/Index_Maintenance to display information about all indexes for Rideshare grouped by tables.

Multicolumn Indexes are considered a specialized index. Most Indexes in Rideshare are single column indexes.

3. <https://www.postgresql.org/docs/current/indexes-multicolumn.html>

A database with a large amount of Multicolumn indexes could indicate excessive or redundant indexing, where single column indexes would serve the same purpose and consume less space. You'll want to look for overlapping indexes for possible optimization. We'll cover that later.

On the other hand, Multicolumn Indexes can provide some of the best performance when each of the columns covered in the definition are used by a query. Columns are used from left to right. The leftmost column in a Multicolumn Index definition can be used by queries that only need that single column. Soon you'll learn how to make sure all columns of Multicolumn Indexes are being used.

On the same PostgreSQL wiki page, use the "Index size/usage statistics" query for the Rideshare database. This query will show usage information.

To simulate some usage of Rideshare, from your Terminal run `bin/rails server` from the Rideshare directory to start the Rails Server. From another Terminal window, run `bin/rails simulate:app_activity` which will send some requests to the Rideshare server.

With that completed, you will now have some Index usage information from queries that were executing from that simulation. The number of scans shows a count of how many Index Scans have been performed using this index. The tuples fetched and tuples read show information on the number of tuples provided by this index. Indexes with no scans and no tuples fetched may be unused or infrequently used indexes.

Creating good Indexes for your database is challenging because it requires knowledge of multiple factors including the data statistics, the query patterns, and the Index types available.

Is the order of how columns appear in the Index significant?

Understanding Index Column Ordering

The order of columns covered by a Multicolumn Index is significant.

Consider a Multicolumn index that covers columns "a" and "b", ordered (a, b) (a then b) from the left to the right. Which types of queries can use this Index? Queries with column "a" and "b" can use this index, or queries with only column "a" can use this index. Queries with only column "b" cannot use this index. Only the first column of a Multicolumn index can be used.

This makes Multicolumn indexes a type of specialized index. They are useful for specific queries that work with multiple fields, when those fields are in

the correct order. Multicolumn indexes are also useful in other scenarios, but only the first column for queries that work with that column.

For a query with column “b” that above Index won’t be used. What if your application needs an index to efficiently filter on column “b”? PostgreSQL recommends creating separate indexes for each of column “a” and “b”.

The documentation page *Combining Multiple Indexes*⁴ demonstrates how multiple Indexes can be scanned for a query, or even multiple scans of the same Index can be used to combine Index data for AND and OR conditions.

For the example above, having single column indexes on “a” and “b” and a query that filters on A and B, would allow PostgreSQL to combine the results of Index scans on each of the single column indexes.

Since Indexes add some latency to all DML (Inserts, Updates, and Deletes) operations, it’s worth avoiding redundant indexes. This implies that having all three indexes mentioned so far, a single column index covering “a”, a single column Index covering “b”, and a Multicolumn index covering “a” and “b” is redundant.

On the other hand, a Multicolumn index that covers two columns where the first column is for filtering and the second column is for ordering, may provide the best query performance. You may wish to have specific redundancy like this when your Multicolumn index is specialized for certain queries, and single columns indexes are both used for others.

What types of Indexes are best for columns with boolean values?

Indexing Boolean Columns

Indexing columns that store boolean values has some pitfalls. The column may have three possible values when nulls are allowed, otherwise it will have only two possible values.

Imagine a column called `is_deleted` that’s false by default, does not allow nulls, and is set to TRUE when the row is soft deleted. The majority of the rows will have the column value of false.

Adding an index to the `is_deleted` column may not be helpful depending on the queries. When querying for false values, which are the majority of the rows, PostgreSQL will likely scan the whole table with a Sequential Scan to filter out rows with true values that are Soft Deleted rows.

4. <https://www.postgresql.org/docs/current/indexes-bitmap-scans.html>

Indexes work best when they are highly selective, meaning they select a small proportion of the total number of rows. If only 10% of the rows have a value of true, then a query that filters on that boolean column for True values will likely produce an Index Scan and perform well.

Now you're ready to work with some of the less common Index types. First up is the Expression Index.

Transform Values with an Expression Index

An Index that covers a column may transform the column value before being stored. When this type of transformation occurs the Index is called an Index on an Expression. This may also be called a Functional Index because a database function is being called, like LOWER().

Imagine that want to search users by email address in a case insensitive way. One solution would be to transform the email address using the LOWER() function on the column before it's stored. Use the temp.schema in Rideshare that you set up earlier for the example you'll find next. If you need to recreate this schema and table, refer to the code example called "insert_users_generate_series.sql" from an earlier chapter. When you've done this, you'll have the temp schema and temp.users table with 10 million rows populated.

Run this SQL statement from psql to insert more rows into temp.users:

```
INSERT INTO temp.users (
  first_name, last_name, email, type, created_at, updated_at
)
VALUES
  ('Jane', 'Example1', 'Jane@example.com', 'Driver', NOW(), NOW()),
  ('jane', 'example2', 'jane@example.com', 'Driver', NOW(), NOW());
```

Email addresses are case insensitive, so you've created duplicate email addresses now in your system. You want know which Jane should get an email, whether it should be jane% or 'Jane%'. This creates a problem and you can use a Unique Index on an Expression to solve this.

First, remove the users you just created by running this statement:

```
DELETE FROM temp.users
WHERE email IN ('Jane@example.com', 'jane@example.com');
```

After improving your data model by validating that you only accept unique case insensitive email addresses, you'll try and add the users back.

To improve your data model, add the following Unique Index that has an expression, calling the LOWER() function on the email column. Run \timing to toggle timing if you wish to see how long it takes to add this index.

```
CREATE UNIQUE INDEX index_temp_users_lower_email_unique
ON temp.users (LOWER(email));
```

By lowercasing the input value for the email column, you'll be able to compare emails in a case insensitive fashion.

Now we can move on from Indexes on Expressions. You'll work with the GIN index type and see how it can be used.

Using GIN Indexes with JSON

GIN stands for “Generalized Inverted Index.” In the B-tree index type, each item points to a specific row. The ctid row identifier points to a physical location on disk.⁵

GIN indexes are different. For a GIN index, each index entry points to all occurrences for a value where each occurrence is in a different location. A GIN index is more like the index found at the back of a book that lists a term from the book, and all the pages where it appears.

When is a GIN index better than a B-tree index? A GIN index can provide faster retrieval when a given column contains multiple values within it.

In a json or jsonb field, many values can be contained within a single column. Data structures like key-value or even nested key-value structures can be stored as JSON compatible strings in a single column.

Thus, a GIN index can be used to index values within the JSON string in ways that don't work with B-tree indexes. A B-tree index on a JSON column only supports equality matching.

Another type of nested data are arrays placed in a column. A GIN index makes it possible to efficiently query items in the array.

Here's an example using Rideshare. Imagine that the business wishes to collect additional metrics on rides. They want to know how many bags each driver is picking up. This might help with dispatching the properly sized vehicle. They might want to know whether Drivers have the music on or not, and whether a water bottle is offered to the Rider. These data points may be represented as strings, booleans, or numbers.

5. <https://www.postgresql.org/docs/8.2/ddl-system-columns.html>

Since it's unclear whether these additional data points are needed for the long term, and since they're unlikely to ever be joined on, you decide to store this data as JSON compatible text in a denormalized document style representation. Another benefit of this approach is that you'll be able to append more attributes to this list over time. On the downside, you're wondering whether you'll be able to efficiently query this data or not when the row counts in this table grow.

To store JSON compatible strings you can choose between column types of `json` or `jsonb`. JSONB supports indexes on the data so you decide to use JSONB.

Additional details about the attributes are:

- `bags_in_trunk` (a number)
- `music_on` (true or false)
- `water_offered` (true or false)

In the snippet below, `\gset` is used to create a *dollar quoted literal* (See: *Defining multi-line strings in psql*⁶).

This is like a Heredoc or multi-line string in Ruby and is used here to better format the JSON string for printing purposes.

After making the variable `json_string` in `psql`, the variable is referenced in the statement `SELECT jsonb_pretty(:json_string);`.

Open `psql` and run this command:

```
SELECT
$$'{
  "ride_details": {
    "bags_in_trunk": 1,
    "music_on": true,
    "water_offered": true
  }
}'$$ AS json_string \gset
```

You've now set the `json_string` variable. Referencing `:json_string` inside the `JSONB_PRETTY()` function below will format it nicely. Besides formatting, this also ensures the JSON is valid.

Run this statement in `psql`:

```
SELECT jsonb_pretty(:json_string);
      jsonb_pretty
```

6. <https://stackoverflow.com/a/31457183>

```

{
    "ride_details": {
        "music_on": true,
        "bags_in_trunk": 1,
        "water_offered": true
    }
}
(1 row)

```

The following SQL file performs a few actions. First you'll create three JSON variables as JSON formatting strings that represent the data source for the additional attributes.

To store the data, you'll modify the Rideshare trips table and add the data column as a new nullable jsonb column. The SQL statement does assume that trips with primary keys 1, 2, and 3 exist. The dollar quoted literal technique and line breaks are used for legibility.

Run these statements from psql:

```

sql/jsonb_trip_details_multi_insert.sql
-- Populate json_string1, json_string2, and json_string3
SELECT $$ '{"ride_details":
{"bags_in_trunk": 1, "music_on": true, "water_offered": false}
}'::jsonb$$
AS json_string1 \gset

SELECT $$ '{"ride_details":
{"bags_in_trunk": 2, "music_on": true, "water_offered": true}
}'::jsonb$$
AS json_string2 \gset

SELECT $$ '{"ride_details":
{"bags_in_trunk": 3, "music_on": false, "water_offered": true}
}'::jsonb$$
AS json_string3 \gset

-- Add the `data` column to the trips table
ALTER TABLE trips ADD COLUMN data jsonb;

-- Insert 3 json blobs from above into the `trips`
-- table, for ids 1, 2, 3
--
-- use UPDATE ... FROM for multi-row update
-- https://stackoverflow.com/a/18799497
UPDATE trips AS t
SET data = c.json_string
from (VALUES
    (:json_string1, 1),
    (:json_string2, 2),
    (:json_string3, 3)
) AS c(json_string, trip_id)

```

```
WHERE c.trip_id = t.id;
```

Query the following Trips rows by id and confirm their data column is populated.

```
SELECT * FROM trips WHERE id IN (1,2,3);
```

With the JSON string data populated, you're now able to query it. Select rows where ride_details is present in data by running the following statement in psql. Running the query with EXPLAIN shows that a Sequential Scan of the trips table is being used.

```
EXPLAIN SELECT * FROM trips WHERE data ? 'ride_details';
```

With the data populated, you're now ready to Index it and take advantage of the GIN index type. Create the following GIN index by running the statement in psql and then Analyzing the table.

```
CREATE INDEX ON trips USING gin(data);
```

```
ANALYZE trips;
```

Run the query again with EXPLAIN and notice that a Bitmap Index Scan on trips_data_idx is now used instead of a Sequential Scan.

Now try performing an aggregation. Get the average number of bags in the trunk on trips. This might help inform you whether to send cars with bigger trunks at certain times of the day.

```
sql/avg_bags_in_trunk.sql
```

```
SELECT AVG((data->'ride_details'>'bags_in_trunk')::integer)
FROM trips WHERE data ? 'ride_details';
```

Next you'll look at two more indexing options as alternatives to the GIN index you've seen so far.

PgAnalyze describes a containment query (See: *Understanding Postgres GIN Indexes: The Good and the Bad*⁷) example that can be adapted to Rideshare. First add the index, specifying the jsonb_path_ops class in the index creation statement.

```
CREATE INDEX trips_data_path_ops
ON trips USING gin(data jsonb_path_ops);
```

Now, you can answer questions about trips using a containment query like “Which drivers offered a water bottle to Riders?” or “Which trips had two or more bags in the trunk?”

7. <https://pganalyze.com/blog/gin-index>

```
sql/query_json_operators.sql
-- water was offered
EXPLAIN SELECT * FROM trips
WHERE data @> '{"ride_details":{"water_offered": true}}';

-- 2 bags in the trunk
EXPLAIN SELECT * FROM trips
WHERE data @> '{"ride_details":{"bags_in_trunk": 2}}';
```

You'll continue to expand your usage of these new analytics about trips.

Correlate these ride analytics with particular drivers and locations to help answer Rideshare operational questions. To do that you'll explore B-tree indexes with JSON data. B-tree Indexes can be used with JSON data by creating an expression as you did earlier.

Run the following statement in psql to create a B-Tree Index on the data column for expressions on specific attributes. In the case below, the index narrows down all trips where two or more bags in the trunk were recorded.

```
CREATE INDEX ON trips USING btree(data)
WHERE (data->'ride_details'->'bags_in_trunk')::int4 >= 2;
```

With that index in place, the following query can be run efficiently. Confirm the index you just added was used by prepending EXPLAIN. Run \d trips to confirm the index names correlate to the Indexes you expect.

```
EXPLAIN SELECT * FROM trips
WHERE (data->'ride_details'->'bags_in_trunk')::int4 >= 2;
```

When deciding on JSONB with Ruby on Rails for row storage, consider using the store_accessor functionality for the Active Record model described in *Hidden Gems: ActiveRecord Store storage*.⁸

One trade-off to be aware of with Store Accessor is that it's limited to a single level of depth. The above structure where bags_in_trunk is nested under ride_details wouldn't work because of more than one level of depth. For that, use the JSONB capabilities directly from Active Record.

You've now seen how you can use GIN indexes for use cases like efficiently looking up values in JSONB columns.

What else can GIN indexes be used for?

Using GIN Indexes for Full Text Search

You can use a GIN index to provide full text search capabilities to Rideshare.

8. <https://www.honeybadger.io/blog/rails-activerecord-store/>

You'll work with LIKE queries and develop a similarity search solution using GIN indexes and trigrams (See: *Postgres GIN Index in Rails*⁹). Trigrams can be captured and a GIN index can be used to accept a user's search input and find similar text. Trigrams will help create matches on similarity. The query planner will use a bitmap index scan with this type of GIN index.

Run the following query from psql that performs an ILIKE query for some rows in the users table. The table should be populated from the data generators and have at least 20 thousand rows in it. If you wish to reset the Rideshare database, run `bin/rails db:reset` and then `bin/rails data_generators:generate_all` from your Terminal.

```
sql/query_ilike.sql
```

```
SELECT DISTINCT first_name FROM users WHERE first_name ILIKE 'henr%';
```

ILIKE is being used for case insensitive similarity searching. Lowercase "henr" is used as the partial matching text. Passing the text "henr" with case insensitivity matches on Henrietta, Henriette, and Henry.

You should get those three results when you run the query above. The query performed a Sequential Scan of the whole table. You can speed that up using a GIN index.

You'll enable an extension and add the index using a Active Record Migration.

The following Migration has not been added to Rideshare. To add it, generate a migration file by running this command from your terminal:

```
bin/rails g migration AddTempUsersGinTrgmOpsIndex
```

In the generated Migration file, copy and paste the implementation body from the example below. The Migration will enable the extension and then add the index.

```
ruby/migration_similarity_search.rb
```

```
class AddFirstNameSimilaritySearch < ActiveRecord::Migration[7.0]
  disable_ddl_transaction!

  def change
    enable_extension('pg_trgm')

    add_index(:users,
              :first_name,
              name: 'users_first_name_gin_trgm_ops_idx',
              using: 'gin',
              opclass: :gin_trgm_ops,
              algorithm: :concurrently)
```

9. <https://blog.kipprosh.com/postgres-gin-index-in-rails/>

```
end
end
```

Run the following command in your Terminal to apply the Migration.

```
bin/rails db:migrate
```

After adding the index, open `psql` for Rideshare and run `\d users` to confirm that you now see a new GIN index covering the `users.first_name` field.

After adding the index and a bit of warm-up, the same `ILIKE` query from above should be much faster! Confirm the index is used by adding `EXPLAIN ANALYZE` in front of your query in `psql`.

Next you'll see another new concept. PostgreSQL helps you explore the similarity and distance between words. Run the query below to see how similar “henr” is to each of the names. The distances between “henr” and the name can also be printed. The `%` operator¹⁰ is being used in the `WHERE` clause to filter the results.

```
sql/query_similarity_distance.sql
SELECT
  distinct first_name,
  similarity(first_name, 'henr') AS similarity,
  first_name <-> 'henr' AS distance
FROM users
WHERE first_name % 'henr';

-- first_name | similarity | distance
-- -+-----+-----
-- Henrietta  | 0.36363637 | 0.6363636
-- Henriette  | 0.36363637 | 0.6363636
-- Henry      | 0.5714286  | 0.4285714
```

If you hadn't guessed it, the query above is also using the new GIN index you created.

Next you'll change things up a bit.

In the previous query a fragment of a word was matched exactly. What about similar words that don't match exactly? For example, a search query from a user may have a typo. For typo errors where the letters are ordered incorrectly, the search would produce no results which is not a good user experience. We all make typos!

As an example, searching “Henrieta” with one `t` instead of two will not produce any results.

10. <https://www.postgresql.org/docs/9.1/pgtrgm.html>

You can do better! Using the GIN index and `gin_trgm_ops` class you’ve already created covering `first_name`, it’s possible to search and filter using the similarity information you just learned about and find matches even for typos.

Try running the following statement in psql. Note that you’re now using the `SIMILARITY()` function as your filter criteria.

```
sql/query_filter_similarity.sql
SELECT DISTINCT
  first_name
FROM
  users
WHERE
  similarity(first_name, 'henrieta') > 0.4;

-- first_name
-- -----
-- Henrietta
-- Henriette
```

Although this query is much slower, it does find names that are similar. This can be improved further using Full Text Search in Rails with PostgreSQL, documented in Rails Guides.¹¹ By storing a list of all possible words, matches can be made using different operators to match queries against word content that is similar.

That covers it for GIN indexes. Next you’ll continue with B-Tree Indexes but see how to use a Partial Index.

Using Partial Indexes

Partial indexes are B-tree indexes that include an expression in the index definition that limits the rows covered. Reducing the rows in the index means the index will be smaller which consumes less space, but also reduces index maintenance overhead at write time. In other words, as data is being written in, if it doesn’t fit into the expression for a partial index, it won’t be included in the index.

A partial index is a specialized index because it’s optimized for certain values within a column, at the expense of not being useful for others.

Here’s an example to help clarify this. Imagine that the Rideshare application uses “soft deletes” when rows are deleted in the app. As a refresher, “Soft Deletes” means rows aren’t actually deleted but are instead updated to make them hidden from the application.

11. https://guides.rubyonrails.org/active_record_postgresql.html#full-text-search

To make the rows hidden, tables get a `deleted_at` timestamp column that's `NULL` by default and then is set when a row is deleted. Any rows with a `deleted_at` value set are hidden.

Imagine that Users can deactivate their accounts and this action populates the `deleted_at` column value. The Marketing department wants to find those users and offer them incentives to come back to the platform.

Create the index again like you did earlier, except this time exclude the deleted users from the index.

Rideshare does not currently have Soft Deletes enabled, so you'll need to add the `deleted_at` column.

Run the following statements in `psql` for Rideshare to add the column, and then add a Partial Index. After that you'll delete about 10 users, and then analyze the table.

```
sql/index_users_deleted_email_partial.sql
ALTER TABLE users ADD COLUMN deleted_at timestampz;

-- Index on deleted users
CREATE INDEX IF NOT EXISTS index_users_deleted_email_partial
ON users
USING btree (email)
WHERE deleted_at IS NOT NULL;

-- delete some users
UPDATE users
SET deleted_at = NOW()
WHERE id <= 10;

ANALYZE users;
```

Now try running this query:

```
SELECT email FROM users
WHERE deleted_at IS NOT NULL;
```

Prepend `EXPLAIN ANALYZE` to the query and confirm the Index is being used.

An alternative would be to use a Multicolumn index. Add an index covering the `deleted_at` column to filter down to only deleted rows, and then include the email column as the second column in the definition, so that the email can be fetched from the index without needing a Heap Scan.

Run this in `psql`:

```
CREATE INDEX IF NOT EXISTS users_deleted_email
ON users USING btree(deleted_at, email);
```

Now run `\d users` to confirm you have indexes “`index_users_deleted_email_partial`” that you created earlier, and “`users_deleted_email`” you created just now.

Run the query from earlier again, and confirm the Multicolumn index is used and an Index Only Scan is performed.

Review the trade-offs between these two approaches. Both the Partial index and Multicolumn index made the query much faster. They are both special purpose indexes for certain queries. One trade-off is that they consume different amounts of space and have different maintenance overhead.

Consider the size of the Multicolumn index. Run these `SELECT` statements from `psql` to compare the sizes:

```
SELECT
pg_size_pretty(pg_total_relation_size('users_deleted_email'));
pg_size_pretty
-----
1360 kB
```

Now check the size of the partial index:

```
SELECT
pg_size_pretty(pg_total_relation_size('index_users_deleted_email_partial'));
pg_size_pretty
-----
16 kB
```

The partial index is much smaller at 16 kB, but it may be used by fewer queries. You will have to test both of these in your database, carefully analyzing the query performance and the space consumption for each index type.

You’ve now seen how Partial Indexes can be used as specialized indexes for specific queries, with a small file size.

Next you’ll work with a different Index type, the BRIN index, to see how that Index can be used for large result sets while trying to minimize space consumption.

Using BRIN Indexes

You’ll use BRIN indexes to help improve query performance on a large data set. BRIN stands for Block Range Index.

What is a block range index entry?

A block range index entry points to a page (the atomic unit of how PostgreSQL stores data) and stores two values: the page's minimum value and the maximum value of the item to be indexed.¹²

BRIN indexes can be a good fit for analytical queries or queries that use time ranges where the data has a strong correlation between the physical ordering of the rows and how the data is queried.

Since the BRIN index stores page positions and since all rows are stored in pages, the most benefit from a BRIN index for query performance is when entire pages can be skipped.

A BRIN index only stores page values so it consumes little space. Since BRIN indexes are small, they may be worth having in addition to B-tree indexes to help for certain queries.

You saw what a block range index entry is, but what about a block range?

A block range is a group of pages that are physically adjacent in the table

In the post *When does BRIN win?*,¹³ the author shows examples of queries where the data is highly correlated, and when it has low correlation. The query uses an `AVG()` aggregate across a large data set and compares the performance with a BRIN index and a B-tree index including query times and index sizes.

In *Large Data Performance With Minimal Storage*,¹⁴ the author again compares `AVG()` performance on large sets of data using an B-Tree Index and a BRIN index. The BRIN index not only outperforms the B-Tree Index but is massively smaller.

“for BRIN to be effective, you need a table where the physical layout and the ordering of the column of interest are strongly correlated.”

To adapt this example to Rideshare, imagine a Trip Positions model that captures GPS points from Drivers and Riders on trips. This is a high volume insert-only table where rows are not updated or deleted but might be queried.

For this example, you'll insert Trip Position records with a generated series ticking every one second over several months. This ticking second value will be used as the value for the `created_at` column, and will move chronologically through time from the beginning of the range to the end of the range.

12. <https://www.crunchydata.com/blog/postgresql-brin-indexes-big-data-performance-with-minimal-storage>

13. <https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win>

14. <https://medium.com/geekculture/postgres-brin-index-large-data-performance-with-minimal-storage-4db6b9f64ca4>

This creates a lot of data, over 5 million rows to cover that time period in seconds. The rows are considered to have a “high correlation” because the values for “created_at” are clustered together with how they’re stored on disk and they’re similar in time. Low correlation would be values that are similar in time but scattered all over.

This high correlation meets the guideline above for a BRIN index to be effective.

From psql, run this SQL statement:

```
sql/trip_positions_populate_correlation.sql
INSERT INTO trip_positions (
    position, trip_id, created_at, updated_at)
SELECT POINT(' (37.769233' ||
    FLOOR(RANDOM() * 10 + 1)::TEXT ||
    ', -122.3890705)'),
(SELECT MIN(id) FROM trips),
seq,
seq
FROM generate_series(
    '2023-08-01 0:00'::timestampz,
    '2023-10-01 0:00'::timestampz,
    '1 second'::interval)
AS t(seq);
```

With that data created, you’re ready to query it. Run the following query in psql to get counts by day, grouping hundreds of thousands of records by day from the first 5 days of the month.

```
sql/trip_positions_by_day.sql
SELECT
    date_part('day', created_at) AS date,
    COUNT(*)
FROM trip_positions
WHERE created_at >= '2023-08-01' AND created_at < '2023-08-06'
GROUP BY 1
ORDER BY 1;
```

Prepend EXPLAIN ANALYZE and confirm that the query uses a Sequential Scan of the trip_positions table.

The WHERE clause above can use an index on the created_at column. To improve the speed of WHERE clause filtering, use a B-Tree or BRIN index. As you saw in the earlier posts, the B-Tree and BRIN indexes may offer comparable performance but the BRIN index will be much smaller.

To create the BRIN index, use the USING brin clause. Run this statement from psql to add the index and then run ANALYZE trip_positions:

```
sql/trip_positions_create_index_btree.sql
```

```
CREATE INDEX trip_positions_created_at_idx
ON trip_positions (created_at);

ANALYZE trip_positions;
```

With the B-Tree Index in place, the query runs much faster. The query execution plan shows an Index Only Scan using the “trip_positions_created_at_idx” index. Check the size of the index.

Running `SELECT pg_size_pretty(pg_relation_size('trip_positions_created_at_idx'));` in `psql` shows a size over 200 MB. Drop the Index and try creating a BRIN index instead.

```
sql/trip_positions_create_index_brin.sql
```

```
CREATE INDEX trip_positions_created_at_idx_brin
ON trip_positions USING brin (created_at);

ANALYZE trip_positions;
```

Run the same query again. You should see a Bitmap Index Scan using the “trip_positions_created_at_idx_brin” and very good query performance comparable to the B-Tree Index. Check the size of the BRIN index. The size is just around 40 kB. Much smaller!

You’ve seen how to use Indexes for filtering operations. How else can they be used?

Using Indexes In Less Common Ways

Besides speeding up queries, Indexes can be used for sort operations where `ORDER BY` clauses appear in a query. Indexes store sorted data as opposed to how the data might be ordered in pages.

B-tree indexes are sorted in ascending order by default.¹⁵

In this section you’ll put this concept into practice. For Rideshare the trips table has a `completed_at` column that’s set when a trip is completed. The trip’s `completed_at` is `NULL` when a trip is in progress.

Run the following query from `psql` to order the Trips by their completion time. If you’ve run the data generators you should have at least 1000 trip records.

```
EXPLAIN (ANALYZE) SELECT * FROM trips
WHERE completed_at IS NULL
ORDER BY completed_at;
```

The query uses a Sequential Scan of the trips table.

15. <https://devcenter.heroku.com/articles/postgresql-indexes#b-trees-and-sorting>

Taking inspiration from the Heroku “B-trees and Sorting” post from earlier, create an index on `trips.completed_at` that will be used by the `ORDER BY` clause from the query above. This Index orders the `NULL` `completed_at` records at the end. Run this statement in `psql` to create the index:

```
CREATE INDEX trips_completed_at_index
ON trips(completed_at DESC NULLS LAST);

ANALYZE trips;
```

Run the query again. The query is now much faster and the execution plan lists the index “`trips_completed_at_index`”. An Index Scan was performed for the Sort Operation with the Sort Key: `completed_at`.

You’ve now seen a less common way to use an Index. Indexes usage might be more common, including to cover multiple columns as a Covering Index. How does that work? Read on to learn more.

Using Covering Indexes

When the fields being requested match exactly with the fields covered by an index, this type of Index is called a “Covering Index.”

From PostgreSQL version 11 the `INCLUDE`¹⁶ keyword can be used for Covering Indexes to enable Index Only Scans. The `INCLUDE` keyword is a little different from a Multicolumn index but they are two ways to create a Covering Index.

With the `INCLUDE` keyword, columns can be added to a special payload area. Payload columns are not filtered on in a `WHERE` clause but are available if chosen in the `SELECT` part of a query.

In Active Record it’s less common to explicitly select columns. Consider places in your codebase that don’t select specific columns. Look at the minimum set of columns needed, and what columns are already Indexed. You may want to add missing Covering Indexes for performance sensitive areas that select a few columns.

When columns match a Covering Index, PostgreSQL can perform the very efficient Index Only Scan type and get all of the data needed from the Index.

It’s time to create examples of these scenarios to better understand them.

Create a Multicolumn index on `first_name` and `last_name` and then analyze the table, by running these statements from `psql`:

```
CREATE INDEX users_fname_lname_multi_idx ON users (first_name, last_name);
```

16. <https://www.postgresql.org/docs/11/indexes-index-only-scans.html>

```
ANALYZE users;
```

Querying only for the first_name “Elroy” uses the index. Run the following query in psql.

```
EXPLAIN (ANALYZE) SELECT first_name, last_name
FROM users WHERE first_name = 'Elroy';
```

This query uses an Index Only Scan using the users_fname_lname_idx Multicolumn Index.

An alternative to the Multicolumn index would be to use the INCLUDE keyword and set up a payload column. Drop the Multicolumn Index and create the following index from psql. Notice how this is a single column index on the first_name, but the INCLUDE keyword adds the last_name as a payload column.

```
CREATE INDEX users_fname_include_lname_idx
ON users (first_name)
INCLUDE (last_name);
ANALYZE users;
```

With this index, an equivalent Index Only Scan is used with index “users_fname_include_lname_idx”.

If you checked the sizes of these two Indexes you’d see the sizes are about the same. If the sizes are the same, what’s the benefit of one approach over the other? When enforcing uniqueness, the newer Covering Index style with the INCLUDE keyword has some benefits over the Multicolumn Covering index style.

When uniqueness is defined on the first_name column without the INCLUDE payload column, you’d need to create two indexes for the users table. One index would be a UNIQUE constraint Index and the other would be a Multicolumn Covering Index.

With the INCLUDE style, a single Index be used both for adding a UNIQUE constraint and to act as a Covering Index. Try creating the Index INCLUDE style one from earlier and make it a Unique Index.

Now you’ve learned about a lot of the types of Indexes available in PostgreSQL and how to use them for certain types of queries. You saw earlier how Indexes are useful to help speed up queries, but also consume space and thus shouldn’t be created unless they’re needed.

Part of operating an efficient database is identifying Unused indexes and removing them. Managing unused indexes is part of the overall database

maintenance operations that you'll learn how to perform. It's time to dive into common database maintenance challenges and see how to solve them.

High Impact Database Maintenance

You’ve seen how row modifications are made up of tuples, how there can be live and dead tuples, and how this is part of the Multiversion Concurrency Control (MVCC¹) system.

With this knowledge, you’re ready to learn about database maintenance in greater depth and see how automatic maintenance is performed and where you can help out by manually conducting maintenance.

For databases with hundreds of millions of rows and a high volume of queries you’ll need to tune your system and possibly conduct manual maintenance operations.

You’ll learn about the “VAR” aspects of database maintenance, which are Vacuum, Analyze, and Reindex.

Maintenance Terminology



- Visibility Map — Tracks which row versions (tuples) are visible to transactions
- Vacuum — Mark space as reusable, reclaim space (Vacuum Full), update Visibility Map
- Analyze — Update statistics about table data
- Reindex — Rebuild an index

First you’ll get an introduction or refresher on Autovacuum.

1. <https://www.postgresql.org/docs/current/mvcc.html>

Basics of Autovacuum

Automatic Vacuum, or *Autovacuum*, is a background process that runs by default when PostgreSQL starts. Autovacuum can be tuned using a variety of parameters. On modern production servers the default Autovacuum parameter values are considered to be very conservative. For this reason, Autovacuum tuning guides often recommend changing the values to make Autovacuum run more frequently.

If you're completely new to Autovacuum, it can be thought of as the “garbage collection process”² for PostgreSQL. In a roughly similar way to how garbage collection runs periodically in garbage-collected programming languages when objects are deallocated, Autovacuum runs periodically for dead row versions in a database that are no longer needed for any transactions.

Visible Rows are made up of row versions behind the scenes. Row versions can be live, reflecting what's currently visible to all live and running transactions, or dead. When a row is updated or deleted in PostgreSQL it's not actually updated or removed physically, but instead a new row version called a tuple reflects the changes from the update or delete.

Run the following query from psql. This query shows the ctid column, which is a hidden column. The two values displayed are the page number and tuple number.³

```
SELECT ctid,id FROM users WHERE id = 1;
```

Update the same User by running `UPDATE users SET first_name = 'Jane' WHERE id = 1;`

Now run the same query showing the ctid from above. The tuple version will change. The page number may be the same unless a lot of updates have happened, and in that case PostgreSQL may place the row version in a new page.

When a former row version is no longer referenced by any live transaction, it becomes dead. At that point the dead row version may be removed. Until it is removed, dead row versions are commonly referred to as *bloat*.

The way Autovacuum works in depth and the dozens of tunable parameters are beyond our scope. The basic way it works and a couple of tuning parameters will help you get started.

2. <https://www.postgresql.org/docs/current/routine-vacuuming.html>

3. <https://nidhig631.medium.com/ctid-field-in-postgresql-d26977de7b58>

Tuning Autovacuum Parameters

Autovacuum runs Vacuum workers per table when threshold values are met. These threshold values are configurable and there are a lot of them for different purposes.

Generally speaking, you may tune Autovacuum so that it runs jobs more frequently, for longer periods of time, and more of them in parallel. The tradeoff is that by running more workers for longer, you'll use more server system resources. These are the same server resources available to your application query workload, so you'll need to carefully make adjustments and monitor the impact.

As your database gets busier with higher rates of UPDATE and DELETE operations, dead row versions accumulate faster. Autovacuum may fall behind and not clean up dead tuples quickly enough before going to sleep again.

When Autovacuum is not keeping up you'll see a symptom of long VACUUM durations where it might run for hours or even days. VACUUM run times are logged in the `postgresql.log`.

The threshold to trigger an Autovacuum worker for a table is controlled by the parameter `autovacuum_vacuum_threshold`. This can be adjusted by tuning the `autovacuum_scale_factor`.

The default value for the scale factor is 0.2. This means that when 20% of the table size is made up of dead tuples VACUUM is triggered.

Many resources recommend reducing `autovacuum_scale_factor` so that VACUUM is triggered when there are fewer dead tuples. A value of 0.01 (1%) or even less will cause VACUUM to run much more frequently, which may be necessary based on the rate that dead tuples accumulate. As you make these changes, first make the change to specific tables that have a high volume of Updates and Deletes before making the change globally for all tables.

To get more familiar with this, you'll begin to tune Autovacuum for the Rideshare database.

The parameter `autovacuum_vacuum_cost_limit`⁴ specifies a “cost” of work that Autovacuum completes before it stops running. Raising this integer value to specify a higher cost causes Autovacuum to run longer, completing more work.

4. https://postgresqlco.nf/doc/en/param/autovacuum_vacuum_cost_limit/

In PostgreSQL 15 the default value is -1 and a separate parameter `vacuum_cost_limit` is used instead. The `vacuum_cost_limit` default value is 200, and this parameter is in effect when `VACUUM` is run manually from `psql`. Try running `VACUUM VERBOSE` trips; from `psql`. The `VERBOSE` option is optional but shows more information about what's happening.

Override the default by setting `autovacuum_vacuum_cost_limit` to a higher value like 2000. With this value, Autovacuum vacuum jobs will complete ten times the work before stopping. This is a good change to try if your Autovacuum Vacuum jobs are not cleaning up dead tuples at a fast enough rate. This might be worth checking mainly for tables with a very high volume of Updates and Deletes. This parameter value can be changed without restarting PostgreSQL so you can adjust it up and down to observe the effect.

Change the value globally in your PostgreSQL config and reload it using `pg_ctl reload` from your terminal.

```
sh/postgresql_conf_parameter_change.sh
pg_ctl reload -D /path/to/postgresql.conf
server signaled
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "autovacuum_vacuum_cost_limit" changed to "2000"
```

Although it can be changed globally, a better method to start might be to change it only for tables with heavy Updates and Deletes. To do that, as an example run `ALTER TABLE vehicles SET (autovacuum_vacuum_cost_limit = 2000);` from `psql` for the Rideshare database, to adjust the value for the vehicles table.

Another parameter to tune for PostgreSQL versions earlier than 15 is the `autovacuum_vacuum_cost_delay`. This parameter controls the pause duration. A lower value means Autovacuum is paused for shorter periods of time. In the table below the value is reduced from 20ms to 2ms. In PostgreSQL 15 the default value for this parameter became 2ms, so this parameter change isn't necessary there.

Parameter Name	Original Value	Tuned Value
<code>autovacuum_vacuum_cost_limit</code>	200	2000
<code>autovacuum_vacuum_cost_delay</code>	20ms	2ms

Table 3—Autovacuum Database Parameters

Tuning parameter values

These Autovacuum parameter changes were proposed by PostgreSQL Major Contributor David Rowley⁵ after analyzing a PostgreSQL 10 database with

5. <https://www.postgresql.org/community/contributors/>

significant bloat problems. The database had tables with very high amounts of Updates causing dead tuples to accumulate quickly. These changes helped Autovacuum run more effectively.

Another example from Sentry.io in the post *Transaction ID Wraparound in Postgres*,⁶ they describe lowering `autovacuum_vacuum_cost_delay` down to 0ms because Vacuum was not running frequently enough.

You've now seen how to adjust Autovacuum. Shift your focus to the Indexes on your tables to make sure they're maintained well.

Rebuilding Indexes Without Downtime

Periodically rebuilding an index in PostgreSQL ensures it only has pointers to live tuples. As a refresher, Indexes are secondary data stores that are owned by tables.

To rebuild Indexes in PostgreSQL you use the `REINDEX` statement. Dropping and creating the index again achieves the same outcome as it builds the index based on live tuples. But on a live system, this is unsafe because it means the index won't be available for active queries that use it.

On older versions of PostgreSQL, rebuilding indexes while the server was running was not possible. An index could be safely rebuilt (Reindexed) if the database was taken down without disrupting any queries.

Fortunately PostgreSQL has made it possible to rebuild indexes while the server is running. The index remains available to live queries that are using it.

If you're running PostgreSQL version 12 or older you won't yet have access to this capability. Upgrading to PostgreSQL 13 or newer is the best option because there are many improvements besides online index rebuilds. If upgrading is not an option, you've got alternatives. Use the extension and command line program `pg_repack`⁷ to perform online index rebuilds.

The post *Using 'pg_repack' to Rebuild Indexes*⁸ provides an overview of how to use `pg_repack` to replace Indexes online. Briefly, `pg_repack` can be used to rebuild an index. The technique it uses is to rebuild a replacement index and then swap in the replacement while dropping the old index in a transaction. `pg_repack` works well for that purpose but it is a separate client program from

6. <https://blog.sentry.io/2015/07/23/transaction-id-wraparound-in-postgres/>

7. https://reorg.github.io/pg_repack/

8. <https://andyatkinson.com/blog/2021/09/28/pg-repack>

PostgreSQL. What if there was a built-in way to rebuild indexes without disrupting any queries that are using the Indexes?

Fortunately from PostgreSQL 13 onward, you can use the `CONCURRENTLY` keyword with the `REINDEX` command to accomplish this, rebuilding Indexes online without requiring a third party tool like `pg_repack`.

Try putting this into action. From `psql`, run the following command to reindex the `index_trips_on_driver_id` index. The `Verbose` is used below and it's optional but shows more information about what's happening.

```
REINDEX (VERBOSE) INDEX CONCURRENTLY index_trips_on_driver_id;
```

Instead of Reindex with an Index, try specifying a Table to reindex all the indexes for the table. Run this statement from `psql`:

```
REINDEX (VERBOSE) TABLE CONCURRENTLY trips;
```

Running Manual Vacuums

PostgreSQL lays out data on disk in pages that are 8kb in size. The pages are not completely filled by default; space is left available to store new tuples from row modifications. Dead tuples take up space in those pages as well. When Autovacuum runs a `VACUUM` worker for each table in PostgreSQL, one of the jobs it performs is to reclaim the space used by dead tuples. Once completed, the space that was used by those dead tuples is marked as available again for new row modifications.

As a refresher, performing a manual `VACUUM` for a table like `users` is done by running this command from `psql`:

```
sql/vacuum_table.sql
VACUUM users;
```

The `ANALYZE` option can be added, and this updates table statistics for the table. The statistics include estimates that are used by the query planner as you saw earlier.

```
sql/vacuum_analyze_table.sql
VACUUM ANALYZE users;
```

New versions of PostgreSQL continue to add capabilities to Vacuum and Analyze. Version 11 made it possible to specify multiple tables at once. Version 12 added `SKIP_LOCKED` support which skips locked tables (See: *Postgres 12*

highlight - *SKIP_LOCKED* for *VACUUM* and *ANALYZE*⁹). Try running this statement from psql:

```
sql/vacuum_multiple_skip_locked.sql
VACUUM (SKIP_LOCKED) trip_requests, trips;
```

PostgreSQL keeps track of when Vacuum and Analyze are run both automatically and manually.

The following query shows a last_analyzed timestamp for the users table, which is when it had been last vacuumed manually. The timestamp last_auto_analyzed_at shows when the table was analyzed last by Autovacuum.

```
sql/last_analyzed_pg_stat_all_tables.sql
SELECT
    schemaname,
    relname,
    last_autoanalyze,
    last_analyze
FROM pg_stat_all_tables
WHERE relname = 'vehicles';
```

Try running it for the vehicles table. The value for last_analyze may be empty. Run ANALYZE vehicles; and then check the last_analyze value again.

Since PostgreSQL 13, Parallel Vacuum Workers can be configured (See: *Parallel Vacuum in Upcoming PostgreSQL 13*¹⁰).

The default value for MAX_PARALLEL_MAINTENANCE_WORKERS is 2. Try changing it to 4. Specify the PARALLEL option with a value of 4, and set the VERBOSE flag to print more information when the command runs. Up to 4 workers will be started.

```
sql/set_max_parallel_maintenance_workers.sql
SET MAX_PARALLEL_MAINTENANCE_WORKERS=4;

VACUUM (PARALLEL 4, VERBOSE) users;
```

Let's explore Bloat in greater depth.

Simulating Bloat and Understanding Impact

As you saw earlier, Bloat refers to the amount of dead tuples that are present in your system. Dead tuples consume space that could be made available for live tuples. When bloat is very high it can contribute to worse performance.

9. <https://paquier.xyz/postgresql-2/postgres-12-vacuum-skip-locked/>

10. <https://www.highgo.ca/2020/02/28/parallel-vacuum-in-upcoming-postgresql-13/>

Very high bloat is not normally observed in local development databases. Dead row versions might accumulate in a high volume production system with a very high amount of Updates and Deletes and with Autovacuum not tuned and using default conservative values. This problem has lessened in newer versions of PostgreSQL and may continue to lessen in versions in the future.

Bloat management is nonetheless a common challenge, so you'll work with it a bit more in a hands on fashion.

To better understand bloat in your local development PostgreSQL server, you'll disable Autovacuum and create a very high amount of updates and deletes. By measuring the estimated bloat percentage before and after these updates, you can start to see how bloat accumulates.

To get started, collect a bloat estimate of the users table. Use the `pgsql-bloat-estimation/table/table_bloat.sql`¹¹ script from GitHub by running the statement from `psql`. Add the line `WHERE schemaname = 'public' AND tblname = 'users'` just before the `ORDER BY`, to limit the rows to the `public.users` table. The `bloat_pct` shows the current estimated bloat percentage. Take note of the current value.

Next, simulate bloat by creating a lot of Updates to the table. Run the following statement from `psql` to create 10 million updates to the users table with unique values for each update.

```
sql/bloat_simulate_user_updates.sql
UPDATE users
SET first_name =
  CASE (seq % 2)
    WHEN 0 THEN 'Bill' || FLOOR(RANDOM() * 10)
    ELSE 'Jane' || FLOOR(RANDOM() * 10)
  END
FROM GENERATE_SERIES(1,1000000) seq
WHERE id = seq;
```

This statement generates one update per row, matching the Primary Key `id` column to a sequence value. If you've got around 20 thousand rows, you'll see around 20 thousand updates. Each row gets a unique string that's either Bill or Jane with a number added on. Run the statement a few times to generate more updates.

Periodically check the `bloat_pct` column by running the query from earlier. Run some more updates. You'll see the `bloat_pct` growing. With these heavy amounts of updates, you're causing bloat in the table and in the Indexes.

11. <https://github.com/loquix/pgsql-bloat-estimation>

To reclaim the space in the table, you'll need to run `VACUUM FULL` users; from `psql`. Running `VACUUM FULL` on a live system is not recommended because it will lock the table while it runs.

Indexes for the users table will also become bloated and should be rebuilt. This keeps them optimized for queries.

As you saw earlier, you may run `REINDEX` for a single Index, or for all Indexes on a table. Run this statement from `psql` to rebuild one index:

```
sql/reindex_concurrently.sql
REINDEX INDEX CONCURRENTLY index_users_on_email;
```

You've now seen how to use Vacuum, Analyze, and Reindex to help with your database maintenance. What other types of Index maintenance are there?

Removing Unused Indexes

The PostgreSQL Wiki has a dedicated page for *Index Maintenance*¹² worth reviewing.

Indexes are added to tables to be used by queries. PostgreSQL tracks their usage when a query execution plan performs an Index Scan.

To make sure Index Scans are tracked, confirm that the following values are set to on (they are on by default) by running these statements in `psql`.

```
SHOW track_activities;
SHOW track_counts;
```

Regularly search for indexes that aren't used by queries. These are called Unused Indexes and they should be removed. Indexes can consume a lot of space and they also add some latency to write operations as they're maintained.

- Unused Indexes may prevent Heap Only Tuple (HOT) Updates
- Unused Indexes unnecessarily increase the `VACUUM` run time for a table
- Unused Indexes unnecessarily increase query planning time
- Unused Indexes unnecessarily slow down backup and restore operations

Indexes can grow very large, even larger in size than the tables themselves. This can be very surprising at first. These very large Unused Indexes weigh down your system needlessly.

To remove an index safely on a live system, use the `CONCURRENTLY` keyword as you've done many times in earlier examples.

12. https://wiki.postgresql.org/wiki/Index_Maintenance

Before dropping Unused Indexes, spend some time to double check that they aren't used.

- Run the unused indexes query, identifying indexes with no Index Scans
- As a rollback plan precaution, copy the CREATE INDEX definition prior to removal so that it's available to recreate if needed.
- Perform the index removal using the CONCURRENTLY option

If you're removing an Index via an Active Record Migration, perform the index removal conditionally in case the index doesn't exist for all databases where the migration runs. This can be the case if you have many deployments of your application each with their own database, and the database definition has drifted.

To collect the Index definition as a CREATE INDEX statement, run the following query from psql with an example Rideshare table index.

```
sql/index_create_statements.sql
SELECT indexdef FROM pg_indexes
WHERE indexname = 'index_users_on_email';
```

An Active Record Migration that conditionally removes the index might look like this:

```
ruby/remove_index_concurrently.rb
class RemoveIndexUsersEmail < ActiveRecord::Migration[7.1]
  disable_ddl_transaction!

  def change
    if index_exists?(:users, :email)
      remove_index(:users, :email, algorithm: :concurrently)
    end
  end
end
```

With Active Record, if the index cannot be found using the table and field name, the name option can be used to explicitly name the index.

Besides Unused Indexes, another form of Index Maintenance is to prune Duplicate or Overlapping Indexes. What do those look like?

Removing Duplicate And Overlapping Indexes

Indexes cover one or more columns, and PostgreSQL does not prevent creating multiple indexes covering the same columns as long as the indexes have unique names. Try this out in psql by running these statements:

```
CREATE INDEX index_users_on_email2 ON users (email);
CREATE INDEX index_users_on_email3 ON users (email);
```

If you encounter duplicate indexes, one of them can be removed.

Watch out for Index Duplicates

PostgreSQL Does Not Prevent Duplicate Indexes



- Duplicate Indexes can be created to cover the same fields as long as the indexes have different names.
 - Remove duplicate Indexes
-

Exact duplicates are straightforward and not common. What is less straightforward and more common is overlapping or redundant Indexes.

In this scenario two or more indexes cover the same column and only one of them is necessary. The redundant index may have a different definition, but never be used by a query.

For example, when a query filters on the `first_name` column of the `users` table, a single column index covering `first_name` could be used. A Multicolumn index added later covering `first_name` and `last_name` could be used for filtering on `first_name` as well. These two indexes overlap, both covering the `first_name` field. If the Multicolumn index was added for a query later, it's possible the single column index is no longer needed.

You'll want to see if there are Index Scans for each of these Indexes. To help find these, you can use the PgHero tool. PgHero detects and displays overlapping indexes. The source code has a method `duplicate_indexes` that will compare columns being covered by indexes for a table and check for overlapping definitions.

This information is presented in a section on the main screen called *Duplicate Indexes*. For example, PgHero will explain the following.

These indexes exist, but aren't needed. Remove them with a migration for faster writes.

```
users_first_name_idx (first_name) is covered by users_first_name_email_idx (first_name, email)
```

In the example above, the second Multicolumn index can be used by queries for the `first_name` column. PgHero is suggesting that the first single column index on `first_name` is unnecessary. An Active Record Migration is even generated to perform the removal.

Bloat accumulates in tables with high amounts of Updates and Deletes. What about tables that only receive Inserts?

Removing Indexes On Insert Only Tables

For tables where rows are only inserted and not queried (SELECT), updated, or deleted, these kinds of tables may not have a need for indexes.

For Rideshare, `trip_positions` is one example of a table where rows are only inserted. To confirm this, run this query from `psql`:

```
sql/check_table_dml_behavior.sql
-- Check number of inserts, updates, deletes
-- for `trip_positions` table
SELECT
    relname,
    n_tup_ins,
    n_tup_upd,
    n_tup_del
FROM pg_stat_user_tables
WHERE relname = 'trip_positions';
```

If you find Indexes on *Insert Only Tables* and they aren't enforcing a constraint, they may be unneeded and can be removed.

Indexes for PRIMARY KEY and UNIQUE constraints are needed.

For Insert Only tables, since they did not accumulate dead tuples from UPDATE and DELETE operations, in the past PostgreSQL did not VACUUM these tables. Because VACUUM didn't run, table statistics weren't being updated which then could cause incorrect query planning estimates.

That oversight was fixed in PostgreSQL 13 which now performs statistics collection from Autovacuum even for Insert Only Tables.¹³ This was mentioned in the *Row Estimates* episode of the Postgres.fm podcast.¹⁴

You've now seen how Automatic Vacuum jobs should run. What if you want to run manual vacuum jobs or schedule them; how might you do that?

Scheduling Jobs Using `pg_cron`

`pg_cron`¹⁵ is a third party extension that has wide support on cloud hosted PostgreSQL databases. You'll work with it in your local PostgreSQL server.

Note that this is the first time you're working with a different database. As a refresher, connect to PostgreSQL using "psql" from your Terminal and run `\l` to list all the databases. From `psql`, connect to `postgres` by running `\c postgres`.

13. <https://www.cybertec-postgresql.com/en/postgresql-autovacuum-insert-only-tables/>

14. <https://postgres.fm/episodes/row-estimates>

15. https://github.com/citusdata/pg_cron

With that in mind, you're ready to add `pg_cron`. Add `pg_cron` to the extensions list in `shared_preload_libraries` in your PostgreSQL config file. The extension version will need to be compatible with your version of PostgreSQL.

Configure `pg_cron` to use the `postgres` database in the config file. In `shared_preload_libraries`, add `cron.database_name = 'postgres'` to do that.

To schedule a job, the `cron.schedule` function is provided from the extension. This function schedules jobs only in the database where `pg_cron` is configured. At this writing, `pg_cron` can be configured in only one database. Instead of that function, use the function `cron.schedule_in_database` to work around that limitation. This function is more complex but allows scheduling of jobs in other databases besides where `pg_cron` runs. You'll want to use this other function so that you can schedule jobs for the Rideshare database.

For a very high `UPDATE` rate table, you've decided to schedule a periodic manual `VACUUM`. This way `VACUUM` will run more frequently compared with relying solely on Autovacuum. Note that you can adjust Autovacuum settings per table and that's a good place to start. For this example, assume that you've done that already and you'd still like to run a manual Vacuum.

To schedule `VACUUM` for trips in the `rideshare_development` database for 10:00 UTC, run this statement from `psql`:

```
SELECT cron.schedule_in_database(
    job_name:='rideshare trips vacuum',
    schedule:='0 10 * * *',
    command:='VACUUM ANALYZE trips',
    database:='rideshare_development',
    username:='postgres'
);
```

You've now scheduled a job from the `postgres` database where `pg_cron` is running, to run for a table in the `rideshare_development` database, from the `postgres` database.

To view the job that was just scheduled, run this query from `psql`:

```
SELECT * FROM cron.job ORDER BY jobid;
```

Since it's scheduled but has not yet run, there will be no results from this query.

To break down the parts, `cron` is the schema where the function is defined, `job` is the table, and this table is in the `postgres` database.

Besides creating jobs, jobs can be altered like tables. To alter a scheduled job, use the `cron.alter_job` function.

Now that you've created a scheduled job, you'll want to monitor it. Read on to find out how to do that.

Monitoring pg_cron Scheduled Jobs

To see past runs of scheduled jobs, run this query from psql:

```
SELECT * FROM cron.job_run_details;
```

Jobs can be altered. Alter the job schedule to make it run every minute. After running it every minute, there should now be past run details to view in cron.job_run_details.

Run the following query to find the jobid for the *Rideshare trips vacuum* named job scheduled earlier in this section.

```
SELECT jobid, jobname, schedule, command FROM cron.job ORDER BY jobid;
```

If jobid is 1, run the following statement to modify the schedule attribute for jobid 1, setting the cron expression to mean “every 1 minute.”

```
SELECT cron.alter_job(job_id:=1,schedule:='* * * * *');
```

Once this is set, you should see log entries like these:

```
2022-12-28 14:33:00.003 CST [91464] LOG: \
cron job 1 starting: VACUUM ANALYZE trips
2022-12-28 14:33:00.049 CST [91464] LOG: \
cron job 1 COMMAND completed: VACUUM
```

Run the query `SELECT * FROM cron.job_run_details` from psql again in the postgres database to confirm that the status of those runs shows as succeeded.

To verify this further, you should see that PostgreSQL has tracked that the trips table has been Vacuumed.

Check timestamps for the last manual ANALYZE and VACUUM on trips by running this query:

```
SELECT schemaname, relname, last_analyze, last_vacuum
FROM pg_stat_all_tables
WHERE relname = 'trips';
```

You've now configured a VACUUM ANALYZE job to run as a cron job on a schedule.

What if you were able to monitor pg_cron Scheduled Jobs via PgHero?

In a Scheduled Jobs PR,¹⁶ a new *Scheduled Jobs* page was added that adds visibility to `pg_cron` jobs. Scheduled Jobs that are scheduled and jobs that have run are displayed.

One final topic for maintenance is to learn about some helpful maintenance tooling.

Conducting Maintenance Tune-Ups

Performing database maintenance to keep operations optimal is part of your job as a database administrator.

The most important operations you've conducted are `VACUUM`, `ANALYZE`, and `REINDEX` (See: *Maintaining a PostgreSQL database health with ANALYZE, REINDEX, and VACUUM commands*¹⁷).

You also learned why and how to remove unused Indexes and duplicate or redundant indexes.

You saw the `active_record_doctor`¹⁸ gem and `database_consistency`¹⁹ gem; these can also be used to help with database maintenance. Database Consistency has various Checkers including a `RedundantIndexChecker` that helps you find redundant indexes to remove.

You also saw the `rails-pg-extras`²⁰ gem; it can be used for maintenance as well. This gem checks for Duplicate Indexes, Unused Indexes, and Bloat. The gem also looks for what it calls Null Indexes, which are indexes with a high proportion of NULL values. These may be good opportunities to replace a regular B-Tree index with a smaller Partial Index.

Performing maintenance regularly and monitoring the automatic maintenance will help you keep your database running smoothly.

Now you're ready to look at handling higher levels of concurrent activity.

16. <https://github.com/andyatkinson/pghero/pull/3>

17. <https://andreigridnev.com/blog/2016-04-01-analyze-reindex-vacuum-in-postgresql/>

18. https://github.com/gregnavis/active_record_doctor

19. https://github.com/djezzl/database_consistency

20. <https://github.com/pawurb/rails-pg-extras>

Handling Errors From Increased Concurrency

In this chapter you'll see some common challenges faced with higher levels of concurrent use of the database. More concurrency comes with higher query volume. Your database will begin to have many more client connections and this will make it more critical that queries run quickly. You will also likely to see more errors due to lock contention or queries that are canceled, including cancellations due to safeguards you add.

Managing database connections is an important part of managing your server resources. Database connections are a limited resources and as the query volume increases you may begin to use all of the available connections and even need more connections than what's available.

You'll see how to do that in this chapter. You'll add safeguards that help prevent bad queries from consuming too many resources. You'll see how to better use the limited set of database connections that are available.

Although you may not have these issues today, by developing awareness and practicing how to deal with them in advance, you'll be well positioned to solve them if they arise.

Concurrency Terminology



- Connection Pooler — Middle layer software between Rails and PostgreSQL, to more efficiently use database connections
 - Server Connections — Database connections
 - Client Connections — Active Record pool client connections
-

Monitoring Database Connections

Your Ruby on Rails application is a client application from the perspective of a PostgreSQL server instance. When you connect to it with `psql`, that's another client application.

SQL queries that are generated from Active Record code are sent to PostgreSQL using a database connection. Your application uses TCP over a network to communicate with the server. These are the basics, and a deep dive into networking out of our scope.

The total number of connections that can be served concurrently is limited based on the CPU and server resources of your PostgreSQL instance. Remember that PostgreSQL runs as a server process on a single server (computer) and it will use as much CPU, Memory, and Disk resources as it can based on how you've configured it.

In this section you'll take a high level look at how Active Record Ruby code is transformed into SQL statements and sent to the server.

1. You write Active Record Ruby code, chain methods together, there may be lazy evaluation, and eventually one or more SQL queries are produced as query text
2. The “driver” code ensures that the generated SQL query is PostgreSQL compatible
3. In Ruby on Rails the `pg` Ruby gem is a common adapter for connecting to PostgreSQL
4. Active Record has an application side Connection Pool, which is a pool of database connections. A connection is checked out from the pool in order to send a query to the server. The pool size expresses the maximum possible amount of connections that could be in use, although fewer could be in use at any given time.
5. Finally, the SQL queries are sent over the network using a connection. PostgreSQL computes a result, and in a normal scenario it's returned using the same connection. Other possibilities are that a statement does not complete successfully or is canceled.

Connections are opened on demand. The Active Record connection pool is a client and is responsible for closing the connection. If the connection is not closed it will be held open in an idle state indefinitely.

An idle connection can be in two states. It can be performing a transaction (a query happens inside an explicit or implicit transaction), or it can be idle and not performing a transaction.

Most of the time, the state of connections observed will be idle or active. In PostgreSQL you may see idle in transaction,¹ which means the transaction has been opened with BEGIN but is not completing work.

Take a look at the Active Record Connection Pool. In the example below the pool is configured with pool: 5, meaning up to 5 connections can be used.

```
default: &default
  adapter: postgresql
  pool: 5
```

How else can you see what's happening? Query pg_stat_activity to view connection status information. The results include unrelated PostgreSQL background processes, so add a filter so those aren't displayed.

Filter on the application_name. An example application name is /bin/rails, and backend_type could be client backend.

Run the following query in psql. You may want to run the Rideshare server by running bin/rails server in a Terminal window, and then simulate activity by running bin/rails simulate:app_activity in another terminal window. While the simulation is running, this query will display results:

```
sql/list_connections_and_queries.sql
SELECT
  pid,
  datname,
  username,
  application_name,
  client_hostname,
  client_port,
  backend_start,
  query_start,
  query,
  state
FROM pg_stat_activity
WHERE pid != PG_BACKEND_PID() -- exclude this query
AND datname = 'rideshare_development' -- specify app DB
AND state IS NOT NULL; -- exclude PG background processes
```

In your production systems, look for a connection state of idle. The query field has the SQL statement itself. Compare backend_start and query_start. In *Monitoring Stats*² the backend_start is set when the backend process was started, and query_start is when the query started.

1. <https://www.postgresql.org/docs/current/monitoring-ps.html>
 2. <https://www.postgresql.org/docs/current/monitoring-stats.html>

When the query is executing, the state is “active”. When the query is no longer executing, a `state_change` timestamp is recorded. For example, when a connection is no longer active and is now idle, `state_change` should be set at the point where that transition occurred.

An idle connection is available for re-use by the client once it is no longer performing a transaction.

If connections are idle, how can those be managed?

Managing Idle Connections

The `idle_timeout` timeout in the Active Record Connection Pool can be set to determine a maximum allowable time that idle connections can be checked out from the pool. When they are idle and checked out for longer than the max allowed value, those connections will be forcibly returned to the pool. By default the value is 300 seconds (5 minutes).

A similarly named PostgreSQL server parameter exists called `idle_session_timeout`³ which is new in Version 14. The parameter has this description:

Terminate any session that has been idle (that is, waiting for a client query), but not within an open transaction, for longer than the specified amount of time.

This parameter can complement the `idle_timeout` parameter in the Active Record Connection Pool. The server parameter provides an additional layer of protection within PostgreSQL from connections that are idle and not completing any work. The reason those are bad is because they lock up one of the limited database connections.

Idle connections that are open for a long time may be terminated. For example, for pid 87581 with a idle connection and a query that's not actively running, cancel it using a query like this one:

```
sql/pg_cancel_backend.sql
-- cancel pid 87581
SELECT pg_cancel_backend(87581);
```

In *Waiting for PostgreSQL 14 – Add `idle_session_timeout`*, experiments are conducted with `idle_session_timeout` to help demonstrate how it works.

A query is sent to the background and then the `postgresql.log` shows the terminated connection with error FATAL: terminating connection due to idle-session timeout after the maximum time was exceeded. In the second example, despite

3. https://postgresqlco.nf/doc/en/param/idle_session_timeout/

exceeding the maximum `idle_session_timeout` value, a transaction opened with an `BEGIN` but idle was not terminated.

Per *How to Monitor PostgreSQL Connections* from EnterpriseDB, clients are responsible for closing connections.

It is worth mentioning that if the connection is not explicitly closed by the application, it will remain available, thereby consuming resources — even when the client has disconnected.⁴

Setting Active Record Pool Size

The pool size should be set with care to not exceed the maximum number of connections PostgreSQL can handle. The `max_connections` value should reflect a safe maximum possible value. Active Record will use *up to* the pool size of connections. Connections will be held open and idle for up to 5 minutes by default.

The article *Concurrency and Database Connections in Ruby with ActiveRecord*⁵ from Heroku shows how to use an environment variable to control the value for pool to make it easier to change on the fly.

From *Finding and killing long running queries on PostgreSQL*,⁶ run the following query to list queries that have been running for more than 5 minutes. View the duration column to see how long they've been running. Run the following query from `psql`. There will need to be active queries to display any results.

```
sql/long_running_queries_duration.sql
SELECT
  pid,
  NOW() - pg_stat_activity.query_start AS duration,
  query,
  state
FROM pg_stat_activity
WHERE (
  NOW() - pg_stat_activity.query_start
) > INTERVAL '5 minutes';
```

Using the Active Record Connection Pool

Active Record provides an application-level Database Connection Pooling (See: *Database Pooling*⁷) capability.

4. <https://www.enterprisedb.com/postgres-tutorials/how-monitor-postgresql-connections>

5. <https://devcenter.heroku.com/articles/concurrency-and-database-connections>

6. <https://medium.com/little-programming-joys/finding-and-killing-long-running-queries-on-postgres-7c4f0449e86d>

7. <https://guides.rubyonrails.org/configuring.html#database-pooling>

Database connections can be *physical* or *server* connections; server connections are used by the client Ruby on Rails application with the Active Record Connection Pool when it opens connections.

When a proxy server is between Ruby on Rails and PostgreSQL, connections from the client application to the proxy server are called *client* connections. Client connections between Rails and the proxy are re-used. Multiple client connections may use a single physical connection. Server connections between the proxy and PostgreSQL can be used more efficiently by client applications. You haven't yet worked with a proxy server but you'll see more about that soon.

Active Record lazily allocates physical database connections, meaning they're opened as needed and then kept open in an idle state when a query completes.

As a brief refresher on the backend pids and PostgreSQL processes, for new database connections the master PostgreSQL process is forked. Process forking is fast, but does add latency to the client-server round trip time. To eliminate this latency, the application layer connection pooler will keep connections open but idling. This way the process forking latency is not incurred for every new query. This is kind of like leaving your car running when you go into a store for just a minute!

In the next section, you'll look at dedicated connection pooling software that enhances the capabilities beyond what's possible with the Active Record Connection. These capabilities help your application reach higher levels of concurrency beyond what would otherwise be possible.

Running Out Of Connections

Ruby on Rails application instances can be scaled horizontally by running more instances of the (stateless) application. Commonly, background job processing with Sidekiq⁸ is run with multiple processes as well, and these background workers also require database connections.

For puma or Sidekiq, more processes are added with multiple threads of execution, and each thread requires a database connection.

Client queries may overrun the maximum number of database connections. This is called *exhausting* the connections. When this happens, no new queries can run because no connections are available. The server will return an error to the client like FATAL: sorry, too many clients already.

8. <https://github.com/mperham/sidekiq>

The PostgreSQL parameter `max_connections`⁹ can be used to control the maximum number of connections in use. Changing this value requires restarting PostgreSQL.

A default value of 100 reflects a smaller PostgreSQL database server. A modern production PostgreSQL server might have a higher value like 500 or more.

The steps to simulate connection exhaustion are:

- Set the `max_connections` to a low value of 3
- Open 3 `psql` connections to the server, and run a long query
- Open a 4th connection, observing that the connection fails to open because all connections are in use

Run this statement in `psql`:

```
sql/show_max_connections.sql
-- default: 100
SHOW max_connections;
```

Now change it. Edit `postgresql.conf`, setting `max_connections` to 3. Edit the value of `superuser_reserved_connections` to be 1 less than the value of `max_connections`, which is a requirement from PostgreSQL for this exercise. Restart PostgreSQL:

```
sh/postgresql_conf_edit_max_connections.sh
# use `vim` or another text editor
vim /path/to/postgresql.conf

# set to low value
max_connections = 3
superuser_reserved_connections = 2

# restart PostgreSQL, required when changing `max_connections`
pg_ctl restart -D /path/to/data_directory/
```

Now that you've edited `max_connections` to be 3, when you attempt to run a query using a 4th connection you'll see an error. To simulate this run the following script from your Terminal by copying and pasting it into a file and then running it with "`sh exhaust_database_connection.sh`".

```
sh/exhaust_database_connections.sh
#!/bin/bash

# set max_connections to 3
query="SELECT pg_sleep(30)"

for run in {1..4}; do
  echo "running query ${query}. times: ${run}"
  psql |
```

9. https://postgresqlco.nf/doc/en/param/max_connections/

```
--dbname rideshare_development \
-c "$query" & # separate processes
done
```

If you've configured PostgreSQL as indicated above, the 4th query requires a database connection and requesting one produces an error because no connections are available. You'll see output like the following.

```
sh/exhaust_database_connections_output.sh
```

```
running query SELECT pg_sleep(30). times: 1
running query SELECT pg_sleep(30). times: 2
running query SELECT pg_sleep(30). times: 3
running query SELECT pg_sleep(30). times: 4
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed \
FATAL: sorry, too many clients already
```

Running out of database connections is a severe problem.

- In an application with increased use, more database connections will be needed
- Your organization may not wish to scale up (vertically) the database server, which does add CPUs and makes it possible to serve more connections, but costs more money
- Your application may keep a large amount of idle connections open, using up connections
- Idle connections from a long running query use up connections

For a Ruby on Rails application, on the conservative side, queries should be finishing in less than 1 second. For an application in a scaling mode under heavy concurrency, queries should run in the tens of milliseconds.

Run these queries to explore the active connections:

```
sql/idle_transactions.sql
```

```
-- e.g.
-- count,state
-- 7,active
-- 3510,idle
SELECT count(*), state
FROM pg_stat_activity
GROUP BY 2;

--- https://dba.stackexchange.com/a/39758
SELECT * FROM pg_stat_activity
WHERE (state = 'idle in transaction')
AND xact_start IS NOT NULL;
```

How many connections do you need? Conduct some capacity planning and resource utilization investigation.

- How many Ruby on Rails application instances are running? How many threads are configured per process? Each thread acquires a database connection to complete work.
- What is the connection pool size per process? Multiply the number of processes by the configured pool size.
- How many Sidekiq processes are running? How many threads are running per process? Sum the total number of threads to get the total concurrent number of transactions that are possible.

Complete this type of capacity planning analysis, totaling up web processes, background processes, and looking at their pool sizes, to gain a possible maximum theoretical number of concurrent connections the application would use.

Compare that figure with how you've configured `max_connections`. Under provision, leaving some connections available for administrative work like a `psql` session or other PostgreSQL CLI programs like `vacuumdb`. This also assumes there are not other client applications connected to the same PostgreSQL database. As a best practice, give each client application a unique database.

When you're running out of connections, you'll need more help. Fortunately there is additional software you can run for such a scenario.

Working With PgBouncer

When the concurrency of your application grows, the number of physical database connections used by the Active Record Connection Pool alone may not be enough.

PgBouncer, free open source software that you can run in between Ruby on Rails and PostgreSQL, allows you to scale to higher levels of concurrent database activity.

- Client connections may exceed `max_connections`
- Connections can be held open even if the database server is temporarily unreachable, delaying their processing but not refusing the connections
- The pooler re-uses connections more efficiently, achieving higher levels of concurrency on the same compute resources
- Connection Poolers may keep connections open while backends are modified, for example when modifying the server instance

In this section you'll install, configure, and begin using PgBouncer with Rideshare.

Earlier you learned that establishing a database connection has some costs, and adds some latency to the query. Some of the costs are:

- Negotiating SSL
- Authentication

Without reuse of a connection from a pool, these costs are paid for every new query. With a connection pooler, these costs are not paid for each new query or even each new connection. PgBouncer holds connections open to PostgreSQL so that clients like Active Record can reuse them.

On Mac OS, install PgBouncer using Homebrew.¹⁰ Run this command from your Terminal:

```
sh/pgbouncer_install.sh
brew install pgbouncer
```

After PgBouncer is installed, run the following commands to inspect the pgbouncer process and restart it.

```
sh/pgbouncer_commands.sh
brew services info pgbouncer
brew services restart pgbouncer
```

PgBouncer runs on port 6432 which is exactly 1000 higher than the default PostgreSQL port of 5432. Exciting trivia for your next PostgreSQL cocktail party!

Next, set up a PostgreSQL user that will act as the application user to connect with. Run these statements from psql:

```
sql/create_test_user.sql
DROP ROLE IF EXISTS app_user;
CREATE USER app_user WITH PASSWORD 'insecurepwd';
```

Configure the application user. Add the `admin_users = app_user` section below to make `app_user` an admin user. Edit the `.ini` configuration file by opening it in your text editor. Find the file path by running `brew info pgbouncer`. After changing the config file, run `pgbouncer -R /usr/local/etc/pgbouncer.ini` from your Terminal to restart PgBouncer.

```
sh/pgbouncer_ini_config.sh
[pgbouncer]
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = /usr/local/etc/userlist.txt
```

10. <https://brew.sh>

```
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = app_user
```

With PgBouncer configured, you're ready to connect it to the Rideshare database.

Try connecting to PostgreSQL via pgbouncer by running the following `psql` command with a connection string from your Terminal.

```
sh/pgbouncer_psql_connection_string.sh
# pgbouncer user, password, created earlier
# port=6432
psql "postgresql://app_user:insecurepwd@localhost:6432/rideshare_development"
```

PgBouncer acts transparently as a proxy and is compatible with `psql`. View some information about PgBouncer by running these commands:

```
sh/pgbouncer_connect_explore.sh
# Special administration "pgbouncer" database
psql -p 6432 -U app_user pgbouncer

# List commands available from pgbouncer prompt
SHOW help;
```

This brings up a command line interface like `psql` where you can run various commands. PgBouncer can be restarted using a rolling restart. Refer to the documentation for more information.

```
sh/restart_pgbouncer.sh
# Note that -R/--reboot is deprecated. Refer to documentation
# for a rolling restart.
pgbouncer --reboot /usr/local/etc/pgbouncer.ini --daemon
```

PgBouncer supports SQL formatted commands. Try running some of the following commands while connected to the special pgbouncer database.

```
sql/pgbouncer_commands.sql
SHOW help;
SHOW clients;
SHOW databases; -- Review 'pool_size', 'pool_mode', etc.
```

If you ran `show databases;`, one of the options you'll see is the Pool Mode. What's that all about?

Choosing A PgBouncer Pooling Mode

PgBouncer supports various *Pool Modes*. The Pool Mode specifies how a server connection may be reused by clients.

Pool Modes are listed below, ordered from the most aggressive level to the most conservative level in how connections are reused. A more aggressive Pool Mode will reuse connections earlier, potentially achieving higher levels of concurrency, at the cost of having to disable certain client side features.

Some features like *Prepared Statements* and *Advisory Locks* are not compatible with all pool modes. These are covered later.

The descriptions below are directly from PgBouncer.

Mode	Description
session	The default mode. The Server is released back to pool after client disconnects.
transaction	Server is released back to pool after transaction finishes.
statement	Server is released back to pool after query finishes. Transactions spanning multiple statements are disallowed in this mode.

Table 4—Pool Modes

Selecting a Pool Mode for maximum reuse

The *Transaction* Pooling Mode cannot be used with Prepared Statements, which are enabled by default in Active Record.

The statement Pool Mode is not commonly used. This was discussed in the *Connection Poolers*¹¹ episode of Postgres.fm.

The Transaction mode is most commonly used. To use the *Transaction* Pooling Mode, though, Prepared Statements must be disabled.

If disabling Prepared Statements is not a viable path, then the *Session* mode from PgBouncer is the next best option.

Now that you’ve installed and configured a connection pooler, and have an idea of how it can be used, shift your focus to errors related to database connection.

Identifying Connection Errors and Problems

In this section you’ll use psql to analyze connections. Connections may be viewed in PgHero¹² which you began to use earlier.

Some connection errors to avoid are:

- Exhausting connections

11. <https://postgres.fm/episodes/connection-poolers>

12. <https://github.com/ankane/pghero>

- Having too many idle connections
- *Transaction ID wraparound*
- Runaway queries, or zombie queries, that never return results to the client

Some of the safeguards you'll add are:

- Setting a `statement_timeout` to set a max allowed amount of time for a statement to run
- Setting a `lock_timeout` to prevent waiting too long to acquire a lock
- Setting `idle_in_transaction_session_timeout`¹³ to protect against transactions that are idle and not performing any work

You've already seen `statement_timeout` and `lock_timeout` in an earlier chapter.

Focus now on some of the idle timeouts. The following table presents these parameters and some recommended values in milliseconds.

Parameter Name	Default	Tuned Value	Restart
<code>idle_session_timeout (14+)</code>	0	30000	false
<code>idle_in_transaction_session_timeout</code>	0	3000	false
<code>statement_timeout</code>	0	10000	false
<code>lock_timeout</code>	0	3000	false

Table 5—Database Concurrency Parameters

Tuning parameter values

When considering `idle_session_timeout`¹⁴ and `idle_in_transaction_session_timeout`, start by adjusting the latter.

Recommended parameter values are from the *Tightly Coupled* post *GOTO Postgres Configuration for RDS and Self-managed Postgres*¹⁵ and *My GOTO Postgres Configuration for Web Services*.¹⁶

To collect the best values for your particular database, conduct experiments and gradually introduce changes in small increments. Develop a good understanding of how you'll observe the effect and make sure it provides the benefit you're looking for.

Note that Cloud providers will tune some parameters like memory allocation dynamically based on the server instance. Client parameters like timeouts may be left up to customers to enable and adjust.

13. <https://www.postgresql.org/docs/current/runtime-config-client.html>

14. https://postgresqlco.nf/doc/en/param/idle_session_timeout/

15. <https://tightlycoupled.io/goto-postgres-configuration-for-rds-and-self-managed-postgres/>

16. <https://tightlycoupled.io/my-goto-postgres-configuration-for-web-services/>

More Lock Monitoring With `pg_locks`

The `pg_locks`¹⁷ system view provides current information about held locks. In MVCC, all transactions use either a shared or exclusive lock of database resources like a table or rows of data.

You observed lock activity focused on table level locks and concurrent modifications to table structures. Now you'll focus on errors with concurrency from DML operations.

Besides DDL changes like `ALTER TABLE` that use an exclusive lock, Autovacuum uses a `SHARE UPDATE EXCLUSIVE` lock mode. This lock type means that it is both an exclusive lock and a shared lock. Access for reads is Shared, but is locked for updates.

When an `UPDATE` occurs to a row, the row is locked exclusively during the update. Concurrent reads of the row are possible.

`SELECT ... FOR UPDATE` adds additional protection when selecting rows being updated (See: *Selecting for Share and Update in PostgreSQL*¹⁸).

`ROW SHARE LOCK` locks are taken on those rows. Although rows could be selected and updated within a transaction, a concurrent transaction could modify the same rows being selected.

The solution is to open a transaction and select the rows, but add the `FOR UPDATE` keyword to indicate that rows will be exclusively locked for the duration of the `SELECT` statement.

Taking an exclusive lock on rows will block other row `SELECT` operations and modifications. Any concurrent readers or modifications will be in a waiting state for the `SELECT ... FOR UPDATE` transaction block to `COMMIT` or `ROLLBACK`.

The `NOWAIT` keyword can be added to the `SELECT ... FOR UPDATE` to prevent excessive waiting to acquire the lock on the rows to update.

`FOR UPDATE` is considered to create an explicit lock, similar to how the `LOCK` command may be used to explicitly lock a table.

Monitoring Row Locks

Set the `log_min_error_statement`¹⁹ to log queries to `postgresql.log` that may time out due to being blocked on lock acquisition.

17. <https://www.postgresql.org/docs/current/view-pg-locks.html>

18. <https://shroyasha.io/selecting-for-share-and-update-in-postgresql.html>

19. <https://www.postgresql.org/docs/current/runtime-config-logging.html>

SKIP LOCKED limits the selection to rows that are not currently locked.

For a query running repeatedly, skipping locked rows would eventually select all rows, assuming that lock durations are small and the transactions holding them complete successfully.

SKIP LOCKED is a good concurrency control mechanism, when using a database as a queue of items to process.

As mentioned in *Selecting for Share and Update in PostgreSQL*,²⁰ rows on a referenced table by a Foreign Key column will also be locked when SELECT ... FOR UPDATE is used. This is necessary to maintain referential integrity.

Another type of weaker lock is SELECT ... FOR SHARE. This creates a shared lock on the rows being selected. A shared lock allows another concurrent reader to select the same row.

A shared lock still prevents Updates and Deletes of the row, but as a weaker lock type, allows for greater read concurrency.

How do you find lock conflicts?

Finding Lock Conflicts

In the Citus article *PostgreSQL rocks, except when it blocks: Understanding locks*,²¹ SQL operations are put into a table along with lock types and conflicts. When operations conflict with each other, a red X is placed on the intersection. When they do not conflict, a checkmark is placed in the intersecting table cell.

For example, a SELECT will never conflict with a SELECT, so that intersecting table cell gets a green checkmark.

A SELECT *will* conflict with an ALTER TABLE operation, because the ALTER TABLE requires a table level exclusive lock to run, which blocks concurrent reads. Reads are blocked until the ALTER TABLE lock expires which is when the table modification completes.

Each statement acquires a lock type, usually implicitly.

Next, look at conflicts from CREATE INDEX. Scan the CREATE INDEX column, identifying SQL statements that conflict. Any DML operation will require a lock type that conflicts with the lock type a CREATE INDEX takes. This is why creating

20. <https://shiroyasha.io/selecting-for-share-and-update-in-postgresql.html>

21. <https://www.citusdata.com/blog/2018/02/15/when-postgresql-blocks/>

indexes using the `CONCURRENTLY` keyword is recommended for any production database where DML operations are occurring concurrently.

How else can you analyze lock behavior?

Using PgBadger For Lock Analysis

You can again use the PgBadger²² log analysis tool you saw earlier, except this time using the reports it creates for lock analysis.

Using PgBadger is straightforward. Call the executable, passing a single argument which is the path to your `postgresql.log` file.

Run this command from your Terminal:

```
sh/pgbadger_log_analysis.sh
pgbadger /path/to/postgresql.log
```

The program crunches the data and produces an HTML output file that can be opened in your browser for analysis, named `out.html` by default. Run `man pgbadger` to review the large list of optional arguments.

The data presented by PgBadger is historical data from logged activity. Since it relies on log data, it cannot show you what is currently happening.

One of the features of PgBadger is Lock Analysis. Navigate to the top navigation “Locks” tab and explore the information there.

You may want to set a value for `log_lock_waits`²³ and observe the effect. PostgreSQL runs a deadlock checking routine after 1 second.

You may want to monitor for row level locking problems when using `SELECT ... FOR UPDATE`.

What about locking in Active Record?

Active Record Optimistic Locking

In a concurrency context, locks are acquired either optimistically or pessimistically. Pessimistic locks are what PostgreSQL uses, meaning the locks are taken upfront. PostgreSQL supports high levels of concurrent reads and modifications, even with a pessimistic locking strategy.

22. <https://github.com/darold/pgbadger>

23. https://postgresqlco.nf/doc/en/param/log_lock_waits/

When using PostgreSQL with Active Record, an optimistic locking technique can be used. This technique can be used when multiple transactions are updating the same row.

With optimistic locking, a lock is taken “late.” Each process with an optimistic lock can check whether the object being locked has already been modified. When the second modification has lost the “race” to modify the locked object, an `ActiveRecord::StaleObjectError` exception is raised.

To use this functionality, add a `lock_version` column to tables used by Active Record objects that you wish to lock. The column is automatically updated by Active Record when objects are instantiated.

The Engine Yard post *A Guide to Optimistic Locking*²⁴ shows how to handle concurrent modifications. When locking is required for the application, and the locking experience involves user interface level interaction, consider rescuing this error, and rendering the edit page with the latest version of the object instantiated. Add a helpful Rails *Flash Message* for the user, to describe that their edit was not allowed.

When locking is behind the scenes and not user facing, lock management at the row level may best be done using the PostgreSQL capabilities of `SELECT` with either `FOR UPDATE` or `FOR SHARE`. Use `FOR UPDATE` when the goal is to select some rows and prevent concurrent updates.

What about lighter weight locks?

Using Advisory Locks

PostgreSQL provides a weaker lock type called an *Advisory Lock*. Advisory Locks in Active Record use the Ruby standard library `Mutex`.

Ruby on Rails uses Advisory Locks. One example is when Active Record Migrations are run, an Advisory Lock is taken to prevent two threads of execution from applying the same migration. If this happened, an `ActiveRecord::ConcurrentMigrationError` error is raised.

Advisory Locks are useful, but come with other trade-offs. The Transaction Pooling Mode for PgBouncer cannot be used with Advisory Locks. To use Transaction Pooling Mode with PgBouncer, from Rails 6, Advisory Locks can be disabled²⁵ by setting `advisory_locks: false` in `config/database.yml`. As you saw earlier, you’d also need to disable Prepared Statements.

24. <https://www.engineyard.com/blog/a-guide-to-optimistic-locking/>

25. <https://blog.saeloun.com/2019/09/09/rails-6-disable-advisory-locks.html>

Wrapping Up

In this chapter you've started to see how to handle a common challenge in scaling up a PostgreSQL instance which is managing limited database connections. You saw how to identify the number of connections being used and how to scale them up even beyond what PostgreSQL is capable of itself by using a connection pooler.

The upcoming chapters will focus more on Optimizing and Scaling topics.

In the next chapter you'll consider the Ruby code design and Schema design for some common application features, from a scalability perspective, applying what you've learned so far in your evaluation.

Part IV

Optimize and Scale

Scalability of Common Features

In this chapter you'll work outside of Rideshare and analyze the schema design, indexes, and SQL queries for common application features.

First you'll look at a popular gem for adding tags to items. Next you'll look at various pagination techniques and their performance characteristics. These are features that are common to most Rails applications.

When you add Ruby gems as dependencies to your application and they modify your database, you get benefits and possible downsides. The benefit is accelerated development and for popular gems, a solution that's been built and maintained with the input of a lot of developers. The possible downsides are that the schema design, Indexes, or queries may be more general than what you need. You'll know exactly what the tables, queries, and Indexes look like if you get under the hood and take a look. That's what you'll work on now.

Consider the design of a table with Indexes from a gem. There may be an Index to cover all kinds of query scenarios and you may not need all of them. You may have unused indexes you could remove.

Terminology



- **Pagination** — Fetching one page of results for display
- **Keyset Pagination** — Scalable Pagination technique for Indexed columns
- **Database Cursor** — Database object to help with consistent result pagination

How might you go about assessing the design?

Analyzing Schema Designs from Gems

In this section you'll start to create a process you can use for evaluating the database aspects of Ruby gem code. You'll be analyzing the database components of the gem `acts-as-taggable-on`.

This gem is not added to Rideshare. You can add this gem to your local copy of Rideshare or you can generate a separate Rails application to test it where you'll add the gem. Refer to the Development Guides Appendix to see how to generate a new Ruby on Rails application.

Generate a test application where you'll install the gem. To do this, from your terminal run the following commands.

1. Generate a new Rails application using `rails new`. See the Development Guides Appendix for more information.
2. For the newly generated application, add the `acts-as-taggable-on` gem to the Gemfile and run `bundle install`
3. Create the new application database with `bin/rails db:create`
4. Copy the Migration files from the gem source to the application source by running `bin/rails acts_as_taggable_on_engine:install:migrations`
5. Run `bin/rails db:migrate` to apply the Migrations.

For a generated application called `acts_as_taggable_test_app`, the development database is `acts_as_taggable_test_app_development`. Connect to it from your Terminal by running `psql --dbname acts_as_taggable_test_app_development`, or by using `bin/rails dbconsole`.

You're now connected via `psql` or `bin/rails dbconsole` to `acts_as_taggable_test_app_development` database.

Describe the tables with `\dt`. Explore the `taggings` and `tags` table with `\d`.

On the `name` column of the `tags` table there is a single-column Unique index enforcing a unique constraint, and the `name` is nullable. On the `taggings` table, there is a *Foreign Key* constraint from the `tag_id` column to the `tags.id` Primary Key column.

The `taggings` table uses Single Table Inheritance (STI) and has polymorphic associations to a "taggable" type and `id`. The `taggings` table also has a multiple column uniqueness definition across the following six columns.

- `tag_id`
- `taggable_id`
- `taggable_type`

- context
- tagger_id
- tagger_type

One thing that stands out is that the taggings table has a lot of indexes. There are 11 B-Tree indexes, a combination of single and Multicolumn. Some of these indexes may not be necessary.

Now that you know a little about the tables, Indexes, and Constraints for the tables from the gem, what about the queries?

Understanding Queries From Tagging Gem

To analyze queries from the gem, you'll need to add the gem to application models and do either manual or automated testing to generate some activity.

Create a model for the tagging gem.

You decide to do some manual testing. From your Terminal, generate a new Active Record model and open it up with your editor (Vim is shown below).

```
sh/rails generate resource User name:string
bin/rails generate resource User name:string
vim app/models/user.rb
```

Add `acts_as_taggable_on :tags` to the User model.

Now it's time to insert some data. Run `bin/rails console` from your Terminal. Run `ActiveRecord::Base.logger = Logger.new(STDOUT)` to log SQL queries.

First create a User record that you've just configured to add the tagging gem. Run this code in a Rails Console and take note of the queries:

```
ruby/acts_as_taggable_on_loading_tags.rb
irb(main):001:0> User.create!(name: "Andy")

TRANSACTION (0.1ms)  BEGIN
User Create (1.4ms)
  INSERT INTO "users" ("name", "created_at", "updated_at")
  VALUES ($1, $2, $3) RETURNING "id"
  [{"name", "Andy"},
   ["created_at", "2023-01-20 17:04:28.354274"],
   ["updated_at", "2023-01-20 17:04:28.354274"]]

ActsAsTaggableOn::Tagging Load (2.6ms)
  SELECT "taggings".* FROM "taggings"
  WHERE "taggings"."taggable_id" = $1
  AND "taggings"."taggable_type" = $2
  [{"taggable_id", 1},
   ["taggable_type", "User"]]
```

COMMIT

Notice some small details about the generated queries. Within the INSERT transaction the taggings table is queried. This adds some overhead to the Insert. When selecting from the taggings table, an asterisk is used so all fields are fetched in the SELECT statement. This query is unlikely to result in an Index Only Scan because all of the fields would need to be indexed.

The following SQL is a cleaned up version of the SELECT query from above and with parameter values populated. This makes it easy to take this query text and prepend EXPLAIN ANALYZE to look at the execution plan.

`sql/select_from_tags_join_taggings.sql`

```
SELECT * FROM tags
INNER JOIN taggings ON tags.id = taggings.tag_id
WHERE taggings.taggable_id = 1
AND taggings.taggable_type = 'User'
AND (
    taggings.context = 'tags' AND
    taggings.tagger_id IS NULL);
```

To scan the Tags and Taggings table, find out whether Sequential Scans or Index Scans are selected. Open `psql --dbname acts_as_taggable_test_app_development` and run the query with EXPLAIN ANALYZE, reading through the execution plan. The query has multiple filters expressed as AND conditions.

- `tagger_id IS NULL`
- `taggable_id = 1`
- `(taggable_type)::text = 'User'::text`

The database columns evaluated with these filters are `tagger_id`, `taggable_id`, and `taggable_type`.

Since creating the database, adding tables, and adding indexes, VACUUM and ANALYZE may not have yet run for the tables. Run them now.

```
VACUUM ANALYZE users, tags, taggings;
```

Remember that in local development your Indexes will likely not be used when row counts are low.

One trick for local testing is to run `set enable_seqscan = off;` from `psql` to “suggest” that Sequential Scans are disabled. Sequential Scans still may be used but it can be worth trying.

Try disabling sequential scans and running the query above again. After doing that you may now see that Indexes are used for the query. Enable Sequential Scans again by running `set enable_seqscan = on;`.

Now that you've got a User, it's time to add some tags. Open your Terminal and run bin/rails console. Add a tag by running this code:

```
User.find_by(name: 'Andy').tag_list.add('author')
```

Observe the queries that are generated and follow a similar process of identifying the tables, fields, and Indexes that are referenced.

You've now seen one way to add a tagging capability to your Rails application, and how to analyze the database details. Next you'll work with Paginating results of data, a very common requirement.

LIMIT and OFFSET Pagination

In a Ruby on Rails application when viewing a list of items, once there are more than a few dozen items it's common to introduce "Pagination." Pagination means getting pages of results for display, one page at a time. A pagination navigation area might appear on the bottom of a page and present clickable numbered boxes that allow the user to move through pages serially, or jump to a higher page (a bigger offset).

The post *Five ways to paginate in Postgres, from the basic to the exotic*¹ presents different ways to perform pagination. The Ruby gems *Kaminari*² and *Pagy*³ are popular ways to add pagination support to Active Record beyond what it supports by default.

The most common form of pagination is using the LIMIT and OFFSET SQL keywords. Limit and Offset are built in to Active Record with limit() and offset() Ruby methods that generate LIMIT and OFFSET SQL.

Run bin/rails console in Rideshare. Run the following Active Record code and take note of the SQL.

```
ruby/active_record_limit_offset.rb
User.all.limit(10).offset(1)
```

While the LIMIT and OFFSET technique is very common it has downsides. The problems are called "result inconsistency" and "offset inefficiency."

Result inconsistency is when concurrent changes are happening while paginating and either pop into pages when navigating backwards, or shift the page items. This inconsistency might be tolerable depending on how critical it is to have discrete results of pages.

1. <https://www.citusdata.com/blog/2016/03/30/five-ways-to-paginate/>
2. <https://github.com/kaminari/kaminari>
3. <https://github.com/ddnexus/pagy>

Offset inefficiency might be less tolerable. Normally Indexes are used to help query performance. However, as the “Five Ways” post mentioned earlier mentions, indexes may not be used for pagination because no filtering is being performed. This means full table scans and worsening performance.

Row count keeps growing when “soft deletes” are used, which are common. Very high amounts of soft deleted records can worsen LIMIT and OFFSET style pagination because it means more records are being scanned. If the rows were hard deleted or fully removed, sequential scans would be less costly.

“Offset inefficiency” may be acceptable for your application if users do not paginate in very far, or row counts remain low and sequential scans are relatively fast.

When row counts are very high and Limit and Offset pagination isn’t working, you may need to reach for another technique. Let’s look at how database CURSOR style pagination works and how to use it.

Database CURSOR Pagination

PostgreSQL supports a CURSOR keyword which creates a database object. The FETCH command is used to fetch items using a named cursor. Cursor pagination can also be a “style” of pagination but here you’re working specifically with the CURSOR database object.

Cursor pagination with CURSOR objects is not supported in Active Record. Cursor Pagination would make the application stateful and present scalability challenges for multiple running application instances. Stateless web applications are preferred because they are easier to scale.

Client side state is linked to a server side Cursor object which violates a stateless design. It is nonetheless worth exploring the technique here to broaden your understanding of available options. When result consistency is very important and longer transactions are tolerable, a database cursor offers the best consistency among all pagination options.

CURSOR objects are declared and used within an opened transaction block. A CURSOR object persists for as long as a transaction is open.

Cursors are declared with a name. FETCH is used to fetch a set of items in a specified direction.

Try creating a CURSOR object in a transaction that’s kept open. The following statements open a transaction, declare a cursor, then use FETCH to fetch 100

items in the forward direction. Finally, the transaction is closed by being committed, which also removes the cursor.

Open the Rideshare database with psql and run the following statements *individually*. In other words, open the transaction, declare the cursor, and run Fetches individually. When you're done, run COMMIT or ROLLBACK.

```
sql/select_declare_cursor.sql
```

```
BEGIN;
DECLARE users_cursor CURSOR FOR SELECT * FROM users ORDER BY id ASC;
FETCH FORWARD 100 FROM users_cursor;
FETCH FORWARD 100 FROM users_cursor;
COMMIT;
```

Confirm that the first batch id range is from 1 to 100. The second batch is from 101 to 200. Notice how the query orders the results by id ascending.

Database cursors have transaction isolation, so they guarantee consistent results.

Cursor pagination keeps a database transaction open, so it is not compatible with the Transaction pooling mode for PgBouncer you saw earlier. PgBouncer is never compatible with the WITH HOLD statement (See: *PgBouncer Features*⁴).

WITHOUT HOLD⁵ is compatible with Transaction pooling mode. With this type of cursor, it can be used only within a single transaction and is not kept open after the transaction closes.⁶

Although Database Cursors are not supported by Active Record directly and have limitations, they have the best consistency. How might you add Database Cursor style pagination to a Rails application?

The *pagy-cursor*⁷ gem is an extension to Pagy that supports native CURSOR objects.

Additional Cursor types are available like SCROLL but they're out of scope. The post *WITH HOLD Cursors And Transactions In PostgreSQL*⁸ has more information on Cursor types.

That's enough about Database Cursors. It's time to implement a pagination technique called *Keyset Pagination*. Confusingly, Keyset Pagination is sometimes called Cursor Style Pagination. A key distinction is that with Keyset

4. <https://www.pgbouncer.org/features.html>

5. <https://www.postgresql.org/docs/current/sql-declare.html>

6. <https://www.cybertec-postgresql.com/en/with-hold-cursors-and-transactions-in-postgresql/>

7. <https://github.com/Uysim/pagy-cursor>

8. <https://www.cybertec-postgresql.com/en/with-hold-cursors-and-transactions-in-postgresql/>

Pagination there is no server side Database Cursor involved in performing the pagination.

Read on to learn more.

Keyset Pagination

Keyset Pagination has a technical name but is a relatively straightforward concept. With Keyset Pagination, a column is indexed and that same column is used both for filtering with a WHERE clause and sorting with an ORDER BY clause.

To retrieve a page of results, a query orders by the pagination column and sets a max amount of records using a LIMIT clause. To fetch subsequent pages, new queries continue to filter and order on the same column but select rows with a value that's one greater than the previous max value.

Because the column is used for filtering and ordering, the queries can use an Index Scan, keeping it fast even for large row counts.

Query performance is good, but what about consistency?

Keyset pagination is consistent too. Any insertions/deletions before the current page will leave the results unaffected.

What are the downsides of Keyset Pagination?

Downsides are the lack of random access and possible coupling between client and server.

What does the lack of random access mean and what is the coupling between the client and server?

First, let's look at the lack of random access. Using the Users table from Rideshare, use a Keyset Pagination style and the PRIMARY KEY id column to fetch pages of results using SQL. Run this query from psql and confirm that it uses the Primary Key Index:

```
sql/explain_select_users_keyset_pagination.sql
EXPLAIN SELECT * FROM users
ORDER BY id ASC
LIMIT 50;
```

By starting from 1 if a sequence was perfectly sequential, a LIMIT of 50 means you might have rows numbered 1 to 50. The WHERE clause could then start with a condition of id > 50 to start from id 51. However, sequences might have gaps and you may generally not be starting from id 1. The downside with Keyset Pagination is that you may need to move through pages one at a time

in order to retrieve the maximum id value, to then be used as the minimum id value in your next WHERE clause. Hopping to a random specific page is not possible or at least not as easy with this style.

The other downside with Keyset Pagination is coupling between the client and server. The column being filtered on needs to be Indexed. While technically called coupling, if your schema is not constantly adding new columns and filtering on them, and if the column needed for Keyset Pagination is Indexable, the degree of coupling for Keyset Pagination seems acceptable.

To address this downside, you'd make sure filter and order columns are Indexable and you'd add an Index for that column before paginating with it. The index may benefit other filtering queries besides the Keyset Pagination use case.

Next, compare Limit and Offset against Keyset Pagination with large row counts for the Users table and example queries.

From your Terminal, run the script `sh scripts/data_loaders.sh` in the root directory of Rideshare. This script loads 10 million users and then analyzes the table, so it will take a while to load.

With the 10 million user records loaded, try running the following statements in `psql` to view the query execution plans for two alternative queries.

```
sql/select_keyset_pagination_large_minimum_value.sql
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM users
ORDER BY id ASC
LIMIT 50
OFFSET 5000000;

EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM users
WHERE id >= 5000000
ORDER BY id ASC
LIMIT 50;
```

Both queries select all fields from the users table, show 50 results, order by the id column, and start from the value of 5000000 in the rows you'd loaded. When looking at the query execution plans, both queries show an Index scan using the Primary Key index. Why is the Limit Offset style version so much slower?

In the Limit Offset version, PostgreSQL scanned through 5000050 rows of the Index, compared with the Keyset Pagination version where PostgreSQL scanned through just 50.

Besides Index scans, you also added BUFFERS information. The Limit Offset version read 117289 buffers and the Keyset Pagination version read just 2.

Reduced Index scans and Buffer reads are what contributes to a significantly lower-cost query.

You've spent some time so far learning how to make your read queries more efficient. What about your queries that write data?

In the next chapter you'll see how to create more efficient queries that write data into PostgreSQL using Bulk Loading techniques.

Working With Bulk Data

Now you're ready to shift your focus on performance to the write operations going into PostgreSQL. By using Bulk Operations where possible, you'll reduce the resources needed for writes. An example of more efficient write operations is moving from single row inserts or updates, to multi-row inserts and updates.

Besides multi-row DML operations you'll see how to use the PostgreSQL `COPY` command, which can be the most efficient way to load data in bulk.

Ruby on Rails applications, like most web applications, tend to have many more read operations than write operations. In SQL terms, this means there are many more `SELECT`s than `INSERT`s, `UPDATE`s, and `DELETE`s.

With that in mind, why focus on improving writes? One big reason to focus on writes and keep your writes in mind is that vertically scaling a single primary PostgreSQL server has a possible ceiling where a single server can no longer be scaled further. When scaling your server vertically you'll add more CPU, Memory, and Disk resources that will help you achieve a higher rate of writes. Modern cloud providers provide very large and expensive machines to run PostgreSQL so it may be unlikely that you reach the ceiling, but there is a ceiling.

Reads, however, do not have the same ceiling, as they can be scaled horizontally using multiple read replicas. Working with Read and Write splitting is something you'll work on later, including setting up a read only replica. With that in mind, for now you'll continue focusing on write performance.

One of the reasons bulk operations are more efficient is that server connection establishment and database transactions add some latency. With bulk operations, you may be able to consolidate multiple queries that might use multiple transactions and connections, into a single transaction and connection, removing latency.

Besides reducing the run time with improved performance, you'll also free up engineering hours by making your scripts, operations and application queries run quickly. Anecdotally, a script that runs quickly is more likely to be maintained and enhanced over time.

Besides the capabilities in PostgreSQL, you'll work with Active Record and third party Ruby gems to build out a set of approaches for bulk operations.

A scenario called the “thundering herd”¹ is a problem where a high volume of writes suddenly run, in an aberrant pattern from your normal write pattern. For Rails applications, imagine you're using the popular Sidekiq gem for background processing and your PostgreSQL database becomes unavailable. Jobs will continue to be enqueued. When PostgreSQL becomes available again, suddenly there is a high volume of inserts, updates, and deletes to process. Having efficient write operations and spare capacity helps your server survive load spikes.

What other types of writes are common Rails applications?

- Outbound Webhooks that send HTTP requests to third parties and write records of what happened to your database
- User Activity records or core object audit tracking records that are written in a higher volume
- Social media style Metrics like views, likes, and favorites on core objects generating a lot of inserts and updates

Bulk Data Terminology



- Upsert — Update or Insert depending on whether a row exists
- COPY Command — Efficient method for bulk dumping and loading
- Foreign Data Wrapper (FDW) — Connect PostgreSQL to foreign data sources

The scenarios above are common from Social Media to B2B Ruby on Rails applications. Read on to learn about ways to make these operations more efficient.

Creating A Bulk Data Generator Rake Task

As you saw earlier with the Performance Database, having a way to experiment in a database that's separate from production helps you observe behavior

1. https://en.wikipedia.org/wiki/Thundering_herd_problem

and practice techniques. You may wish to fill your Performance Database with an even higher volume of data. You can write scripts in SQL, but for now use use bulk loading capabilities available in modern versions of Active Record.

Active Record added a `.insert_all()` method that makes it easier to insert batches of data efficiently. To get familiar with it you'll explore a Rake task that loads Drivers into the database using `.insert_all()`. The Rake task can be customized to load millions of records.

The task uses the `Faker`² gem to generate unique first and last names.

Place the Rake task definition into the `Rideshare lib/tasks` directory. Run it from your Terminal by typing `bin/rails data_generators:drivers`.

```

ruby/fake_data_generator.rake
require 'faker'

namespace :data_generators do
  desc "Generator Drivers"
  task :drivers, :environment do |t, args|
    TOTAL = 20_000
    BATCH_SIZE = 10_000
    results = Benchmark.measure do
      TOTAL.times.to_a.in_groups_of(BATCH_SIZE).each do |group|
        batch = group.map do |i|
          first_name = Faker::Name.first_name
          last_name = Faker::Name.last_name
          Driver.new(
            first_name: first_name,
            last_name: last_name,
            email: "#{first_name}-#{last_name}-#{i}@email.com",
            password_digest: SecureRandom.hex
          )
        end.map do |d|
          d.attributes.symbolize_keys.slice(
            :first_name, :last_name,
            :email, :password, :type
          )
        end
        Driver.insert_all(batch)
        puts "Created #{batch.size} drivers."
      end
    end
    puts "Analyze table...#{Driver.connection.execute("ANALYZE users")}"
    puts results
  end
end

```

2. <https://github.com/faker-ruby/faker>

Note the following design details:

- Change the total amount constant to load more data, in batches at a time.
- Increase the batch size from 10,000 to write more rows at once.
- For each batch, Driver objects are first initialized in memory.
- For each initialized object, only specific fields are sliced out. These are non-nullable fields.
- The `created_at` and `updated_at` timestamps are set in the database but could also be set in Ruby.

This form of `.insert_all()` uses initialized Driver objects in memory but it's also possible to use primitive types like Arrays and Hashes. Validations on the Active Record instances are not run. For example, the Driver model validates the Drivers License Number and that isn't performed when using this bulk loading technique.

You've now seen a way to perform batched inserts using Active Record. What about when you want to work with batches of records in query code?

Batching With Active Record

Active Record methods `.find_each()` and `.in_batches()` offer a way to retrieve records up to 1000 at a time by default, to set a ceiling on the maximum amount of memory in use at a time. You may already be using these methods because at a certain size of data, it's no longer possible to load all records at once. These methods do not improve your write efficiency, but they're relevant as batch operation methods in a broader sense.

Batches of rows are ordered by `id` in ascending order by default. `.in_batches()`, and `.find_each()` in Rails 6.1 can also be ordered in reverse using an `order: :desc` option (See: *Rails 6.1 supports ORDER BY clause for batch processing methods*³).

Remember that anywhere that you're inserting lots of data in bulk, on the other side of the coin where that data is being loaded and iterated through, you'll want to use batch methods.

Besides Inserting and Selecting data in batches, what other operations can be done in that style?

3. https://www.bigbinary.com/blog/rails-6-1-supports-order-desc-for-find_each-find_in_batches-and-in_batches

Performing an Upsert in Active Record

Active Record provides `.update_all()` methods to update multiple rows with the same value at once. To write simpler code with less conditional branching, you may want to Update data when it exists or Insert it when it doesn't. Combining these operations into one verb is called an *Upsert*⁴ and it's supported in various ways between Active Record and PostgreSQL.

Following the same name conventions as `.insert()` and `.insert_all()`, Active Record supports single Upserts using `.upsert()`, and multi-row Upserts using `.upsert_all()`. By using `.upsert_all()` over `.upsert()` you'll write less code but get the same multi-row efficiency behavior.

PostgreSQL does not natively support an UPSERT keyword, although the SQL MERGE keyword added in newer versions of PostgreSQL provides similar behavior. You'll work with MERGE soon.

With `.upsert_all()` when inserting or updating data, the data being supplied as input may cause conflicts with database constraints. Those constraint violations will need to be handled before the operation can run, or by indicating how you'd like to handle conflicts as part of the statement. In the post *Rails: insert_all and upsert_all*,⁵ the `unique_by` key is used with columns listed that define uniqueness.

PostgreSQL supports ON CONFLICT, which is a clause that can be added to control the behavior of multi-row updates. Active Record uses ON CONFLICT and UNIQUE constraints to identify conflicts and know what to do. One option for conflicts is to add the option DO NOTHING, which means that the Insert or Update will not raise a constraint violation error, but will also not succeed. Another ON CONFLICT option is the DO UPDATE which means that some columns will be updated when constraint violations occur.

Active Record added support for *Upsert* behavior that developers can use with methods `.upsert()` and `.upsert_all()`. In Rails 7, additional options were added to `.upsert_all()`.⁶

An example you might use with Rideshare would be to call the `.upsert()` method on a Trip instance:

```
upsert(attributes, returning: nil, unique_by: nil)
```

4. <https://wiki.postgresql.org/wiki/UPSERT>

5. https://www.johnnunemaker.com/rails-insert_all-and-upsert_all/

6. https://blog.kiprsh.com/rails-7-adds-new-options-to-upsert_all/

You'll break down the options here. First is the attributes acting as the data source for the Insert or Update. Next is the “returning” option, which is using the RETURNING keyword that lets you specify the fields to return on Insert. You're specifying this via Active Record. If none are specified, the PRIMARY KEY id column is returned by default.

The .unique_by() option specifies a column name, multiple column names, or an index name.

What about these options:

- update_only
- on_duplicate

The update_only option can be used to specify one or more columns that will be updated in the event of a conflict.

The on_duplicate option cannot be used at the same time as update_only, but is also intended to specify a list of columns to be updated. This behavior would be triggered when there are constraint violations between the data you're trying to Insert or Update, and the existing row data.

A database UNIQUE constraint must be in place to be enforced, and then it's up to you to use these options to control the behavior.

Let's go into more depth with handling constraint conflicts.

Handling Conflicts With ON CONFLICT

As you saw in the last section, these conflict modes are available options you specify:

- ON CONFLICT DO NOTHING
- ON CONFLICT DO UPDATE

Try working with these using specific examples. Create a people table in the same “temp” schema you've been using. Insert a few sample rows. Run these statements from psql:

```
sql/create_table_people.sql
CREATE SCHEMA IF NOT EXISTS temp;

CREATE TABLE temp.people (id bigint PRIMARY KEY, name TEXT);
INSERT INTO temp.people (id, name) VALUES (1, 'Jane');
INSERT INTO temp.people (id, name) VALUES (2, 'Bob');
```

Since the PRIMARY KEY constraint enforces unique values for the id column, inserting a row with a duplicate id will fail. Run this Insert in psql and verify an error is returned:

```
sql/insert_into_people_duplicate.sql
```

```
INSERT INTO temp.people (id, name) VALUES (1, 'Will Fail');
```

You want to insert the data, but you’ve got a constraint violation to handle. What you do from this point forward is going to really depend on the needs of the application and your goals.

First, try the INSERT specifying the DO NOTHING option. Query the people table afterwards and confirm the original row data has not been modified.

```
sql/on_conflict_do_nothing.sql
```

```
INSERT INTO temp.people (id, name)
VALUES (2, 'Bobby')
ON CONFLICT (id) DO NOTHING;
```

You should see INSERT 0 0 meaning no rows were inserted.

When conflicts occur, PostgreSQL sets up something called EXCLUDED, that you can reference as an alias for the source data including referencing specific fields on it. For example, you could reference EXCLUDED.name even though the id column conflicted, to get the original name value that you were attempting to insert.

For your application, you’ve decided that for this id, you want to overwrite the name. Using EXCLUDED.name and a SET that specifies the name column, run the following statement from psql to perform this overwrite.

```
sql/on_conflict_do_update.sql
```

```
INSERT INTO temp.people (id, name)
VALUES (2, 'Bobby')
ON CONFLICT (id) DO UPDATE
SET name = EXCLUDED.name;
```

You should see “INSERT 0 1” which normally means one row was Inserted, although in this case it was actually an update. Query the people table for id 2 and make sure the name reflects the new value.

Using an INSERT statement with DO UPDATE, you’re effectively performing an *Upsert*. The same statement will either Insert or Update the row data.

Review the table below that summarizes some of the PostgreSQL and Active Record methods you’ve seen so far that are related to bulk operations.

Operation	AR or SQL	Version
.find_all()	AR	
.find_each()	AR	
.in_batches()	AR	

Operation	AR or SQL	Version
.insert_all()	AR	6.0
.upsert()	AR	6.0
.upsert_all()	AR	6.0
RETURNING	SQL	
MERGE	SQL	15.0

Table 6—Bulk Operations Active Record and SQL

Using Bulk operation APIs

You’ve seen some of the capabilities for bulk operations to write and read data that are provided by Active Record. What about capabilities in Ruby that are beyond what Active Record supports?

Beyond Active Record With `activerecord-import`

Prior to Rails 6, the `activerecord-import`⁷ gem was one of the go-to solutions to extend Active Record with support for bulk operations. The `activerecord-import` gem has been added to Rideshare where you can experiment with it.

In newer versions of Rails, Active Record added methods like `.insert_all()` and `.update_all()` and they have lessened the need for `activerecord-import`.

However `activerecord-import` still has additional capabilities and you’ll explore some of them in this section. One of the features is an `.import()` method.

The `.import()` API accepts an array of objects of varying types, including Active Record instances, or simple types like Hash, and String. Validation may be performed optionally, by setting `validate` to true.

```

ruby/activerecord_import.rb
# import from Hashes
vehicles = [
  {name: "Monster Truck", status: 'draft'},
  {name: "Jetski", status: 'draft'}
]
Vehicle.import(vehicles, validate: false)

# import from object instances
vehicles = [
  Vehicle.new(name: "Monster Truck", status: 'draft'),
  Vehicle.new(name: "Jetski", status: 'draft')
]
Vehicle.import(vehicles, validate: false)

```

7. <https://github.com/zdennis/activerecord-import>

The *Columns and Arrays* feature allows passing columns and arrays separately. This could be useful for building an importer.

Activerecord-Import supports a `batch_size` option to control the number of items in a batch being inserted. Normally a single INSERT statement is produced to insert all items at once, but this option gives you the ability to split it up into multiple Insert statements that each still Insert multiple rows at once. Active Record model callbacks will not be called with this approach.

Try out the `batch_size`⁸ option of Activerecord-Import.

Set a `batch_size` to 3 for 6 Vehicle instances, which should generate 2 INSERT statements, each statement containing 3 Vehicles.

From your Terminal, run `bin/rails console` and enter this Ruby code:

```
ruby/activerecord_import_batch_inserts.rb
# Log SQL statements in Rails Console
ActiveRecord::Base.logger = Logger.new(STDOUT)

# Set up 6 Vehicle instances
vehicles = [
  Vehicle.new(name: "Lucid Air", status: VehicleStatus::DRAFT),
  Vehicle.new(name: "Bugatti Veyron", status: VehicleStatus::DRAFT),
  Vehicle.new(name: "Ferrari LaFerrari", status: VehicleStatus::DRAFT),
  Vehicle.new(name: "Porsche 911 GT2 RS", status: VehicleStatus::DRAFT),
  Vehicle.new(name: "Toyota GR Supra", status: VehicleStatus::DRAFT),
  Vehicle.new(name: "McLaren Senna", status: VehicleStatus::DRAFT)
]
columns = [ :name ]

# batch_size of 3, should generate 2 INSERT statements,
# each statement with 3 vehicles
Vehicle.import(columns, vehicles, validate: false, batch_size: 3)
```

If you had conflicting data and wanted to clear out all data, run `Vehicle.destroy_all` beforehand.

In the logged SQL statements, you should see two INSERT statements and multiple Vehicles in a list in the VALUES portion of the statement.

Another caveat with this approach is that Counter Cache columns you saw earlier won't be updated.⁹ You'll need to update those manually after a bulk import. Remember that after bulk operations, it's also a good idea to update table statistics with PostgreSQL which you can do with the ANALYZE command.

8. <https://github.com/zdennis/activerecord-import#batching>

9. <https://github.com/zdennis/activerecord-import#counter-cache>

Now that you've started to perform multi-row Inserts, what other types of multi-row statements are possible?

Performing SQL Multi-Row Operations

You've used the ROW_NUMBER() Window Function to help find duplicates to remove before adding a UNIQUE constraint. Once duplicates were found, a multi-row delete operation was performed.

Besides multi-row deletions, updates can run for multiple rows at once. Take a look at how to do that next.

Create a table of list items and generate 10 rows all with the same list_id, numbered sequentially with no gaps from 1 to 10.

Run these statements in psql, using the temp schema to create tables:

```
sql/list_items_create_and_generate.sql
-- temp schema
CREATE SCHEMA IF NOT EXISTS temp;

-- create list_items table
CREATE TABLE temp.list_items (
    id SERIAL PRIMARY KEY,
    list_id INTEGER,
    position INTEGER,
    CONSTRAINT unique_pos UNIQUE (position)
);

-- generate 10 rows, `id` and `position` numbered 1-10
INSERT INTO temp.list_items (list_id, position)
SELECT 1, generate_series(1, 10, 1);
```

To keep this example simple, the primary key id is also numbered from 1 to 10 since you're creating new rows in a new table starting from 1, which ends up being the same as the position column.

Imagine that you wanted to swap the position values between the first and second row. One way to do that would be two UPDATE statements, updating the first row and then the second row.

However the position has a UNIQUE constraint on it so you can't temporarily create a duplicate position. You could use three UPDATE statements that temporarily re-assign one of the values to an invalid list item position like "-1". With that out of the way, you'd be free to move position 2 into 1, and then move position 1 into 2.

Try running these statements in psql:

```
sql/list_items_swap_multiple_updates.sql
-- Depends on: list_items_create_and_generate.sql

BEGIN; -- perform work in a transaction

-- make position 1 available, to list item 2
UPDATE temp.list_items SET position = -1 WHERE id = 1;

-- set list item 2, to position 1, now that it's available
UPDATE temp.list_items SET position = 1 WHERE id = 2;

-- set list item 1, to position 2
UPDATE temp.list_items SET position = 2 WHERE id = 1;

END;
```

This solution works and swaps the numbers around, but you can do it more efficiently with a multi-row update. Instead of multiple update statements, you'd like to create a single statement. A single statement avoids possible consistency problems that multiple Update statements could cause if they didn't all run as expected.

First, you'll need to temporarily duplicate the position value. You've seen that this is possible but requires deferring constraint enforcement until transaction commit time. Fortunately the Unique Constraint is a constraint type that supports deferred enforcement, so you'll use this trick.¹⁰

From psql, try opening a transaction and making the Unique constraint deferred by running `SET CONSTRAINTS temp.unique_pos DEFERRED;`.

```
SET CONSTRAINTS temp.unique_pos DEFERRED;
ERROR:  constraint "unique_pos" is not deferrable
```

This means the constraint is not configured the way you need it – Unique constraints are not deferrable by default.

To fix this, drop the constraint and then create it again configured as a deferrable constraint. Run these statements to do that:

```
ALTER TABLE temp.list_items DROP CONSTRAINT unique_pos;
ALTER TABLE temp.list_items ADD CONSTRAINT unique_pos UNIQUE (position) DEFERRABLE;
```

You've now added the constraint back and have made it deferrable. Performing this on a live system will take more planning, but since you're performing these changes directly on your local instance there's no risk.

Take a look at the multi-row Update statement you'll run that will swap the position of the first and second item:

10. <https://emmer.dev/blog/deferrable-constraints-in-postgresql/>

```

sql/list_items_swap_single_update.sql
-- Depends on: list_items_create_and_generate.sql

-- use UPDATE... FROM syntax
-- https://stackoverflow.com/a/18799497

UPDATE temp.list_items AS li SET
    id = li2.id,
    position = li2.position
FROM (VALUES
    (2, 1),
    (1, 2)
) AS li2(id, position)
WHERE li2.id = li.id;

```

Using the deferrable Unique constraint, you'll open a transaction, defer the constraint, run the multi-row Update, then commit the transaction. To do that, run these statements from psql:

```

sql/list_items_swap_single_update_transaction_deferred.sql
BEGIN;

SET CONSTRAINTS temp.unique_pos DEFERRED;

UPDATE temp.list_items AS li SET
    id = li2.id,
    position = li2.position
FROM (VALUES
    (2, 1),
    (1, 2)
) AS li2(id, position)
WHERE li2.id = li.id;

COMMIT;

```

You've now updated the rows in a single Update statement and have deferred the Unique constraint until the transaction commit time, ensuring your data remains consistent.

Now you know how to multi-row Inserts and Updates, you've seen Upserts, and you've seen how to perform effective Upserts using the ON CONFLICT clause.

The SQL MERGE command was added in newer versions of PostgreSQL and acts as an Upsert. How does it work?

Upserts With SQL MERGE

PostgreSQL 15 added support for the MERGE command. Now you'll explore how it works.

Besides the MERGE keyword you'll use two new clauses:

- WHEN MATCHED THEN
- WHEN NOT MATCHED THEN

Using the same people table you created before and with the same two records, write a SQL statement like the one below.

In this example the VALUES data aliased by people_data refers to an existing id value that *will* match. Since it will match, the branch WHEN MATCHED THEN will run, which is configured to update the name using the SET command.

```
sql/merge_when_matched_then.sql
MERGE INTO temp.people p
  USING ( VALUES(3, 'Jill') ) people_data
  ON people_data.column1 = p.id
WHEN NOT MATCHED THEN
  INSERT VALUES (people_data.column1, people_data.column2)
WHEN MATCHED THEN
  UPDATE SET name = people_data.column2;
```

When running this command you should see “MERGE 1” as a result. Now try changing the VALUES(2, 'William') clause to have data that does not already exist.

Change it to VALUES(3, 'Jill'). This time the WHEN NOT MATCHED THEN branch is selected causing an INSERT, adding Jill to the people table. The result is still “MERGE 1” but there are now 3 records in the table.

Refer to SQL MERGE¹¹ for more examples.

You’ve now seen how to do Upsert style statements using the newer SQL MERGE command.

Shift your focus now to performing bulk operations using the pg_dump and pg_restore client programs.

Working With pg_dump And pg_restore

Row data from PostgreSQL tables can be dumped into a file by using the pg_dump client program. The same dump file can also be restored back into a table using the pg_restore program. There are a variety of ways to do this, and one option is to dump the rows as INSERT statements. This means these Insert statements can be brought to another database and treated like bulk inserts. The dump file as plain text may be compressed to keep the file size small. This can be an efficient way of doing data copying when there is an exact schema match between different databases.

11. <https://www.postgresql.org/docs/current/sql-merge.html>

Perform a dump and restore for a single table¹² to get familiar with this technique. Dump the users table from Rideshare. The .dump file will be compressed and but can still be restored using `pg_restore`. But when it's compressed it won't be readable as it will no longer be plain text.

Run this `pg_dump` command from your Terminal:

```
sh/pg_dump_users_table.sh
pg_dump |
  --format=c |
  --host localhost |
  --table users |
rideshare_development > users_table.dump
```

Now you can try restoring the file. You only dumped a single table, the users table. However, you can use a throwaway database on your local computer to restore into and it will not have all of the Rideshare tables, only the Users table.

Create a database called `temp_load` using `createdb`. Then run `pg_restore` specifying that database, and the `users_table.dump` file. If you don't have permissions for `createdb` from your Terminal, connect using `psql` and run `CREATE DATABASE` from there.

```
sh/pg_restore_users_table.sh
dropdb --if-exists temp_load;
createdb temp_load;
pg_restore --dbname temp_load users_table.dump
```

Once you've restored the users, run `psql --dbname temp_load`. From `psql`, run `\dt` and confirm only the users table was restored. The users table should have the same row count between both databases.

You've now seen one way to dump and restore a single table. The `COPY` command is another way.

Populating Table Data With COPY

You can load into PostgreSQL from CSV files. The CSV file can contain millions of rows and using `COPY`, you can load them in quickly. `COPY` is often the most efficient way to bulk load data.

Some use cases for the `COPY` command are:

- Consuming a large public data set as CSV files

12. <https://dev.to/andyatkinson/dump-and-load-a-postgresql-table-p0o>

- Ingesting data from another PostgreSQL system using an intermediate CSV or dump file
- Creating an Extract Transform Load (ETL) data pipeline, dumping data into CSV so that it's in a well known format to load into another database

Here you'll use COPY TO to dump data. You'll need an absolute file path that points to the a CSV file that will be made from the query results.

Try dumping data from Rideshare. Connect with `psql --dbname rideshare_development` and run these statements:

```
sql/copy_to_users_table.sql
-- COPY command
COPY users TO '/Users/andy/users_dump.csv'
WITH DELIMITER ','
CSV HEADER;
```

You'll see output like "COPY 20210" indicating 20210 rows were copied. Open the file at the path you specified, which was `'/Users/andy/users_dump.csv'` in the example.

Running `du -h /Users/andy/users_dump.csv` from the Terminal shows that the result file for 20 thousand rows is around 3 MB.

The COPY command can be configured to only dump a filtered set of rows as well. Use a SQL query to filter results that go into the CSV file.¹³

This query dumps half the rows (See: *Split table in (PostgreSQL) randomly 50/50*¹⁴) ordered randomly:

```
sql/copy_command_with_query.sql
COPY (
  SELECT *
  FROM users
  ORDER BY RANDOM()
  LIMIT (SELECT COUNT(*)/2 FROM users)
) TO '/Users/andy/users_dump_half.csv' WITH DELIMITER ',' CSV HEADER;
```

By using Random ordering, this technique could be useful to generate slices of sample data.

Take a look at some of the options. The `WITH DELIMITER ','` option adds a comma delimiter (a separator) between the columns.

CSV is the file type and `HEADER` is used to include a header row in the file, which is the first row listing all the column names.

13. <https://hevodata.com/learn/postgres-export-to-csv/#m1>

14. <https://stackoverflow.com/a/52175450>

PostgreSQL supports the server COPY command that you’ve used so far, and a “client” \copy command. Depending on how your PostgreSQL server is configured, you may need to use the client copy command but fortunately it works about the same.

What are some use cases for exporting CSVs? At most companies, there are a lot of people that have worked with Spreadsheet software like Microsoft Excel or Google Sheets. As an engineer, you may be able to help non-engineer team members by providing a CSV form of data that they can easily import into their Spreadsheet application to help with business processes. Be careful not to export sensitive information.

Besides internal company use cases, some third party APIs provide their data in CSV format. You’re now easily able to import CSV data in bulk.

In the Awesome Public Data Sets¹⁵ GitHub project download a Transportation category data set.

In the uber-tlc-foil-response repo, the file uber-trip-data/uber-raw-data-apr14.csv is around 500MB and has around 500K rows of data in it. This data represents Uber pickups in the New York City area from April 2014. This data is similar data to Rideshare, where Trips are started from a Trip Request, and Riders specify a start location.

You will bulk load this data into PostgreSQL. As a reference, pull up the PostgreSQL Documentation for Using COPY on the *Populating a Database* page.¹⁶

When working with data from files, make the table UNLOGGED since WAL protection isn’t needed, and data can be loaded into Unlogged tables faster.

In this case, you’ll create the trip_pickups within the same transaction as using the COPY command to load the data. Skip data integrity checks. This will allow you to populate the data as quickly as possible.

The column names and database types were selected based on sample rows from the Uber data file.

Create an experiments database to store these tables. This separates this data from the Rideshare database since you won’t be linking it together. Create the trip_pickups table in the experiments database.

Run these statements from psql:

15. <https://github.com/awesomedata/awesome-public-datasets>

16. <https://www.postgresql.org/docs/current/populate.html>

```
sql/create_table_trip_pickups.sql
DROP DATABASE IF EXISTS experiments;

CREATE DATABASE experiments;

CREATE UNLOGGED TABLE trip_pickups (
    time TIMESTAMP WITHOUT TIME ZONE,
    lat NUMERIC(15,10),
    lon NUMERIC(15,10),
    base_code VARCHAR(10)
);
```

Next you'll work with the COPY command. Use COPY FROM instead of COPY TO since you're loading from a file.

To make sure you can access the file, after downloading it take note of the path. From psql, run this statement to make sure the file can be accessed:

```
sql/select_pg_stat_file.sql
SELECT pg_stat_file('/path/to/uber-raw-data-apr14-sample.csv');
```

An error ERROR: could not stat file means the path or filename is wrong. If you've already populated this table and want to clean it out, run TRUNCATE trip_pickups.

You've now confirmed that the CSV data file is accessible, and you've got an empty table in an empty database to load the data rows into.

From psql, run this COPY command:

```
sql/copy_command_copy_from.sql
COPY trip_pickups
FROM '/path/to/uber-raw-data-apr14.csv'
CSV HEADER;
```

This should return COPY 564516 or similar meaning 564516 rows were created.

As discussed earlier, COPY operations in the same transaction as CREATE TABLE are fastest.

COPY is fastest when used within the same transaction as an earlier CREATE TABLE or TRUNCATE command.

Create a transaction that creates the table, performs the COPY command, and ends with a COMMIT. Place these SQL commands in a file called copy_command_trip_pickups.sql:

```
sql/copy_command_trip_pickups.sql
-- Drop the existing table in "experiments" database
DROP TABLE trip_pickups;

BEGIN;

CREATE UNLOGGED TABLE trip_pickups (
```

```

time TIMESTAMP WITHOUT TIME ZONE,
lat NUMERIC(15,10),
lon NUMERIC(15,10),
base_code VARCHAR(10)
);

COPY trip_pickups
FROM '/uber-rlc-foil-response/uber-trip-data/uber-raw-data-apr14.csv'
CSV HEADER;

COMMIT;

```

Now you can run these statements as a file using `psql`, from your Terminal.

The following command runs `psql` with the `experiments` database and loads the statements from the `copy_command_trip_pickups.sql` file using the `--file` option.

```

sh/psql_bulk_load.sh
psql --dbname experiments --file copy_command_trip_pickups.sql

```

You’ve now seen how to dump and load large amounts of table rows as CSV files. Did you know that a CSV can be used directly by PostgreSQL like a table? Read on to learn more.

Creating A File Foreign Data Wrapper (FDW)

PostgreSQL makes it possible to connect one PostgreSQL server instance to another, or work with data in files and treat it like a table. This functionality is called Foreign Data Wrappers (FDW) and there are many that PostgreSQL supports.

It’s time to work with the File Foreign Data Wrapper module¹⁷ (`file_fdw`) to turn CSV files into on-the-fly database tables.

With the CSV file acting as a table, you’ll be able to use SQL queries like “`SELECT * from file_name LIMIT 1`” to pull out results from the file. One big limitation is that the file is Read Only and rows cannot be added to it using this mechanism.

How might this be used in a business use case? Soon you’ll see how, when historical table data can be archived, CSV could be a good format to archive it in. With CSV files acting as the storage mechanism for archived data, it’s still possible to connect to them with PostgreSQL and query them if that is needed.

17. <https://www.postgresql.org/docs/current/file-fdw.html>

To create a table from a file, you'll need a CSV to start with. The PostgreSQL documentation uses the postgresql.log file because every PostgreSQL has one of those, but by default it's not in a CSV format.

Fortunately the postgresql.log file can be turned into a CSV file. Do that first. Open up postgresql.conf and set log_destination to csvlog.

In PostgreSQL 15, JSON was added as a PostgreSQL log format (log_destination = jsonlog). Although the JSON format is not useful for this particular example, JSON formatted logs might be a more realistic real world format compared with CSV or the default log format, for organizations with JSON log ingestion and analysis tools.

Getting back to the CSV logging format, after you've set the csvlog format in your config file, you'll need to restart PostgreSQL. The first two lines below show the config file change. The third line shows how you might restart PostgreSQL using pg_ctl, but you'll need to supply the path to your data directory.

```
sh/postgresql_log_csvlog.sh
# add 'csvlog'
log_destination = 'stderr,csvlog'

logging_collector = on

# restart PostgreSQL
pg_ctl restart -D /path/to/data/dir
```

With PostgreSQL restarted and logging in CSV, you're ready to move to use the new format.

To use the CSV log file, you'll now create a table called pglog that's based on the CSV formatted postgresql.log file. To do that, run the following CREATE FOREIGN TABLE from psql, replacing the filename with the path to your log file. Specify "format 'csv'" and take note of the FOREIGN in the table definition.

```
sql/create_foreign_table_pglog.sql
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
```

```

error_severity text,
sql_state_code text,
message text,
detail text,
hint text,
internal_query text,
internal_query_pos integer,
context text,
query text,
query_pos integer,
location text,
application_name text,
backend_type text,
leader_pid integer,
query_id bigint
) SERVER pglog
OPTIONS ( filename 'log/postgresql-2023-01-05_191254.csv', format 'csv' );

```

If the table creation and File FDW worked correctly, you can now query rows from your log file.

Run this query to list 10 recent queries from the log file:

```
sql/query_pglog.sql
```

```

SELECT
    log_time,
    message,
    detail,
    query,
    query_pos,
    query_id
FROM pglog
WHERE query IS NOT NULL
LIMIT 10;

```

This technique could be used to build a simple log analysis solution that makes it possible to query log files using SQL. Note that `file_fdw` can be used with any CSV file, so you can also try to use it with the Uber data file you saw in an earlier section.

You could also create a SERVER object to encapsulate the connection information to a CSV data file.¹⁸

Using the CSV export from users earlier, try setting up a new CSV file based table with a server. Run the following command to generate a small CSV file with 10 rows and no header row. Omit the `HEADER` command. The rows will only be data rows. Limit the output to 2 fields (`id`, `first_name`) and 10 rows.

18. <https://www.postgresql.org/docs/current/sql-createserver.html>

```
sql/copy_command_users_sample_csv.sql
```

```
COPY ( SELECT id, first_name
      FROM users
      LIMIT 10
    )
TO '/Users/andy/users_sample.csv'
WITH DELIMITER ','
CSV;
```

Next you'll create the `SERVER` object. This object will be used to access the foreign data using `file_fdw`. Run this `CREATE SERVER` statement in `psql`:

```
sql/create_server.sql
```

```
CREATE SERVER file_server
FOREIGN DATA WRAPPER file_fdw;
```

With the `SERVER` in place, create a new `FOREIGN TABLE` `users_ft` that references it.

```
sql/create_foreign_table_file_fdw.sql
```

```
CREATE FOREIGN TABLE users_ft (
  id bigint,
  first_name text
) SERVER file_server
OPTIONS (
  filename '/Users/andy/users_sample.csv',
  FORMAT 'csv'
);
```

Now `users_ft` can be queried like any other table. Although you can't add new rows to it this way, since it's backed by a plain text CSV file, try adding rows to the CSV file using a text editor or Spreadsheet software.

Open `psql` again and query the rows. You'll see the newly added row data is now available in your SQL query result.

Wrapping Up

In this chapter you started working with bulk data in a variety of ways, and then started working with database objects. You put PostgreSQL to work turning a CSV file into a database table.

Next you'll bring in a lot of what you've learned so far to scale up from a single PostgreSQL database instance, to multiple instances. By putting multiple instances to work, you'll meet the growing demands of users for your popular Rails application!

Scaling With Replication And Sharding

Content to be supplied later.

Boosting Performance With Partitioning

Content to be supplied later.

Part V

Advanced Uses

Advanced Usage and What's Next

Content to be supplied later.

Installation Guides

Content to be supplied later.

psql Client

Content to be supplied later.

Getting Help

Content to be supplied later.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head over to <https://pragprog.com> right now and use the coupon code BUYANOTHER2022 to save 30% on your next ebook. Void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit <https://pragprog.com/become-an-author/> today to learn more and to get started.

We thank you for your continued support, and we hope to hear from you again soon!

The Pragmatic Bookshelf



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

<https://pragprog.com/book/aapsql>

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up-to-Date

<https://pragprog.com>

Join our announcement mailing list (low volume) or follow us on Twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

<https://pragprog.com/news>

Check out the latest Pragmatic developments, new titles, and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders: <https://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://write-for-us.pragprog.com>

Or Call: +1 800-699-7764